NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DESIGN AND IMPLEMENTATION OF A PROTOTYPE ONTOLOGY AIDED KNOWLEDGE DISCOVERY ASSISTANT (OAKDA) APPLICATION

by

Ann Y. Lee
Edward C. Powers

December 2006

Thesis Advisor:	Magdi Kamel
Second Reader:	Neil Rowe

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** December 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Design And Implementation of a Prototype Ontology Aided Knowledge Discovery Assistant (OAKDA) Application			**5. FUNDING NUMBERS**
6. AUTHOR(S) Ann Y. Lee and Edward C. Powers			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The World Wide Web (WWW) has become a major source of easily accessible information for students, professionals, researchers and the general public. However, the volume of information available through the Web is so overwhelming that it is not unusual to get tens of thousands of "hits" when conducting a relatively simple search. Most existing search techniques use brute force based on keyword matches to find related Web pages. While the enormous speed of search engines improves the efficiency of such methods, effectiveness is not improved.

The objective of this thesis is to construct and test an ontology-based application to help users identify the most pertinent keywords for a search. By navigating ontologies that describe domains of interest, users are assisted in finding a relevant set of key terms that will aid the search engines in narrowing, widening, or refocusing a Web search. Specifically, the thesis develops an ontology-aided Web search assistant prototype to help users enhance the relevance and precision of the returned results through the use of a context provided by ontologies associated with each search.

14. SUBJECT TERMS OWL AIDED INTERNET SEARCH KNOWEDGE BASE DISCOVERY DESCRIPTION LOGICS ONTOLOGY			**15. NUMBER OF PAGES** 205
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DESIGN AND IMPLEMENTATION OF A PROTOTYPE ONTOLOGY AIDED KNOWLEDGE DISCOVERY ASSISTANT (OAKDA) APPLICATION

Ann Y. Lee
Civilian, Department of Defense
B.S., University of California, Berkeley, 1998

Edward C. Powers
Civilian, Department of Defense
B.S., University of Maryland, 1989

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY MANAGEMENT
and
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2006

Authors: Ann Y. Lee

Edward C. Powers

Approved by: Magdi Kamel
Thesis Advisor

Neil Rowe
Second Reader

Dan C. Boger
Chairman, Department of Information Sciences

Peter Denning
Chairman, Department of Computer Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The World Wide Web (WWW) has become a major source of easily accessible information for students, professionals, researchers and the general public. However, the volume of information available through the Web is so overwhelming that it is not unusual to get tens of thousands of "hits" when conducting a relatively simple search. Most existing search techniques use brute force based on keyword matches to find related Web pages. While the enormous speed of search engines improves the efficiency of such methods, effectiveness is not improved.

The objective of this thesis is to construct and test an ontology-based application to help users identify the most pertinent keywords for a search. By navigating ontologies that describe domains of interest, users are assisted in finding a relevant set of key terms that will aid the search engines in narrowing, widening, or refocusing a Web search. Specifically, the thesis develops an ontology-aided Web search assistant prototype to help users enhance the relevance and precision of the returned results through the use of a context provided by ontologies associated with each search.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

We would like express our sincere gratitude to Dr. Magdi Kamel for his guidance and insight, but most especially his patience, without which this project could not have been completed.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The World Wide Web (WWW) is an easily accessible source of information for students, professionals, researchers, as well as the general public. However, the volume of information available through the Web is so overwhelming that it is not unusual to get tens of thousands of "hits" when conducting a relatively simple search. Improving Web searches has, therefore, become an important and potentially lucrative area of research and development. Since the current Web lacks embedded semantics which allow machines to decipher the true content of the Web pages, as Tim Berners-Lee envisions for the “Semantic Web”, most Web search portals use brute force techniques based on keyword matches mapped to indexed Web content. While the enormous speed of search engines improves the efficiency of such methods, effectiveness is not improved.

The objective of this thesis is to construct an ontology-based application to help users identify the most pertinent keywords for a search. By navigating ontologies that describe domains of interest, users are assisted in expanding the relevant set of key terms that will aid the search engines in narrowing or refocusing a Web search. Specifically, the thesis develops an ontology-aided Web search assistant prototype to help users enhance the relevance and precision of the returned results through the use of context provided by ontologies. An experiment to measure whether the developed application benefits the end user is also conducted as part of this thesis.

This thesis is organized as follows. Chapter II is dedicated to understanding the semantics and syntax of RDF and OWL. Although many ontology development tools hide the details of OWL syntax, an ontology developer must have a good understanding of the language in order to construct a valid ontology. Chapter III describes the proposed methodology for building an ontology. It details a seven-step approach for developing OWL based ontologies. A geography domain is used to illustrate OWL constructs and for developing a complete geography ontology for use in the OAKDA application. Chapter IV discusses the use of onotologies as knowledge bases in the OAKDA application. It specifically describes three example scenarios of using ontologies to

discover domain knowledge and create intelligent Web searches. Chapter V details the architecture and construction of the OAKDA application. This chapter describes in detail the components used to build the prototype application and how they communicate with each other within the architecture of OAKDA. Chapter 6 tests the main thesis research question: Can an ontology-based Web search application increase the effectiveness of Web search results over existing approaches for those searches that require a deep contextual knowledge of the domain of interest? The experiment compares the effectiveness of the results of Web search queries formulated by study participants using the OAKDA application with those obtained by the same participants using the widely popular Google search engine. The results show that OAKDA and the ontologies which it implements have a statistically significant positive effect on the precision of web search queries generated by the participants in the experiment. Finally Chapter 7 concludes the thesis by summarizing our effort to develop a tool to help users improve their Web searches. The chapter discusses the application's effectiveness, the lessons learned, and possible future enhancements.

I. INTRODUCTION

A. BACKGROUND

The World Wide Web (WWW) is an easily accessible source of information for students, professionals, researchers, as well as the general public. However, the volume of information available through the Web is so overwhelming that it is not unusual to get tens of thousands of "hits" when conducting a relatively simple search. Improving Web searches has, therefore, become an important and potentially lucrative area of research and development. Since the current Web lacks, for the most part, embedded semantics which allows machines to decipher the true content of the Web pages, as Tim Berners-Lee envisions for the "Semantic Web", most existing search techniques must use brute force based on keyword matches to find related Web pages. While the enormous speed of search engines improves the efficiency of such methods, effectiveness is not improved.

We define an effective search as one returning to the user a manageable set of highly relevant results. More formally, an effective search returns a results set with a high degree of "aboutness." Aboutness is broadly defined as a degree in which a set of returned resources is "about" a particular domain of interest. For instance, "if a system determines that a document *d* is topically related (i.e. about) to query *q*, then the document is returned to the user." [Bruza et al., 2000, 1] Unfortunately, the current technology's inability to identify the true context of Web sites makes it difficult to determine the aboutness of any resource. However, the aboutness of Web search results can be improved if there is an appropriate set of search terms that narrows the query match results to the most relevant sets of domain resources.

Broad keywords or one-word searches will often result in a large number of hits, many of them irrelevant. Conversely, using the appropriate keyword(s) will result in high degree of aboutness in the returned set, therefore resulting in a highly effective search. In order to take advantage of currently available search engines to return user-specific relevant results, an intelligently crafted list of keywords and phrases needs to be formed. To develop such a list, it is necessary to have sufficient knowledge of the domain of context. Unless the user is a subject matter expert, finding the relevant and

related terms of a domain may be difficult or even erroneous. An ontology, which is model of a domain of context, can fill the role of a domain expert and support the identification of precise and relevant keywords.

B. OBJECTIVES

As discussed, an effective means for obtaining more effective Web search results is to have a comprehensive understanding or knowledge of the context of the search term(s). The availability of a domain knowledge in conjunction with the search terms would be extremely useful for determining an appropriate set of search terms, thus retrieving the most appropriate Web resources. In order to gain contextual knowledge, ontologies, defined as "specification of conceptualization," [Gruber, 1993, 1] can be mined to discover relevant information about the domain of interest. Specifically, this thesis refers to an ontology as a system of knowledge representation that allows machines and humans to understand the definitions of concepts and the relationships between them. The authors propose ontologies as an appropriate knowledge representation system that can be used to create search queries with a greater aboutness value.

The objective of this thesis is to construct an ontology-based application to help users identify the most pertinent keywords for a search. By navigating ontologies that describe domains of interest, users are assisted in finding a relevant set of key terms that will aid the search engines in narrowing, widening, or refocusing a Web search. Specifically, the thesis develops an ontology-aided Web search assistant prototype to help users enhance the relevance and precision of the returned results through the use of a context provided by ontologies associated with each search.

C. THE RESEARCH QUESTION

The primary research question of this thesis is: *Can an ontology-based application be built to narrow, expand, refine and increase precision of Web search terms?* In order the address the primary question, the thesis will also address several secondary questions. First, *what is an appropriate approach for accessing and processing of contextual information of an OWL knowledge base?* Second, *what is the*

most appropriate architecture for the prototype application? Third, *how can an ontology inference engine interface with the application?* Last, *is there a method of visually rendering the ontologies for greater usability, navigation and comprehension of domain knowledge?*

The ability to answer these questions will depend largely on finding and determining the necessary software components and building the interface capabilities between them. The challenge will be to learn the necessary technologies and skills to develop the most appropriate architecture and implementation.

D. SCOPE, LIMITATIONS AND ASSUMPTIONS

The scope of this thesis is to research and understand ontologies, their development languages and methodologies, build, and test an ontology-based Web application that aids users in designing search term lists. While the realization of the Semantic Web would allow search engines or agents to process all Web resources based on their content rather than as string of characters, this vision requires redevelopment of all Web sites currently available. The application proposes an interim solution for Web search by employing ontologies as its knowledge system for discovery and search results enhancement using current Web technologies.

The success of the application is contingent on the availability and reliability of ontologies for domain knowledge representation. Although at the time of this thesis, libraries of ontologies are still limited, the prevalence and popularity of ontologies is growing in various fields, both in academia and commerce. It is the authors' belief that applications, such as the one developed here, further encourage the growth and validity of ontologies and suggest a usable architecture for building applications which operate in this milieu.

It is assumed that the readers have a working knowledge of HTML and XML, as well as have a general understanding of software systems. Readers are also assumed to be familiar with Web search engines, such as Google and Yahoo!, and how Web search is performed using one or more terms. Furthermore, it is the authors' argument that

ontologies' contextual domain knowledge provides the user with a selection of domain-relevant search terms which narrows and improves the search results.

E. METHODOLOGY

The methodology used in the development of this thesis consists of two parts. The first part is dedicated to the research and understanding of ontologies, languages, and methodologies for ontology development. This is achieved by first understanding and mastering existing semantic languages for the Web, namely Resource Description Framework (RDF) and Web Ontology Language (OWL). Second, a review of literature on ontologies is conducted, within and outside the vision of the Semantic Web, emphasizing their important contribution to knowledge representation in information technology. Third, existing development methodologies are adapted to construct OWL ontologies, and a seven-step methodology for constructing valid OWL ontologies is proposed. Fourth, a sample ontology is built to demonstrate the semantics of RDF and OWL, as well as the proposed ontology development methodology. Last, examples are demonstrated to show how ontologies can be used to discover knowledge and aid the construction of relevant search terms.

The second part of the methodology used in this thesis is related to the development of the prototype Ontology Aided Knowledge Discovery Assistant (OAKDA), pronounced "Oak D-A", application that will assist users with discovering domain knowledge and designing their Web search terms for the most relevant results. It is best described as a prototype software development methodology. First, a multi-tier architecture is designed for the application. Second, a reasoning engine is incorporated, using TCP/IP interface model, to inference ontologies. Third, ontology concepts and relationships are graphically displayed and can be traversed in any direction. Last, a user function to select relevant terms from the ontology and build a search term list is constructed and used to query the Google Web search portal and return matched results. In order to test the research question, this thesis will also examine whether ontology-aided Web searches perform better than those relying only on a search engines such as Google. While it is difficult to measure the accuracy or aboutness of search results, the

research design will measure the answer precision and content-relatedness of the returned Web pages of the test questions in order to determine the performance of the ontology-aided search tool developed in this thesis.

F. ORGANIZATION OF THE STUDY

This thesis is organized as follows. Chapter II is dedicated to understanding the semantics and syntax of RDF and OWL. Although many ontology development tools hide the details of OWL syntax, an ontology developer must have a good understanding of the language in order to construct a valid ontology. Chapter 3 describes the proposed methodology of building an ontology. It details a seven-step approach for developing an OWL ontology. A geography domain is used to illustrate OWL constructs and for developing a complete geography ontology for use in the OAKDA application. Chapter 4 discusses the use of ontologies as knowledge bases in the OAKDA application. It specifically describes three example scenarios of using ontologies to discover domain knowledge and create intelligent Web searches. Chapter 5 details the architecture and construction of the OAKDA application. This chapter describe in detail the components used to build the prototype application and how they communicate with each other within the architecture of OAKDA. Chapter 6 tests the thesis research question by comparing the precision and content-relatedness of the ontology-aided searches to those performed solely by search engines. Finally Chapter 7 concludes the thesis by summarizing our effort to develop a tool to help users improve their Web searches. The chapter discusses the application's effectiveness, the lessons learned, and possible future enhancements to the application.

THIS PAGE INTENTIONALLY LEFT BLANK

II. RDF AND OWL ONTOLOGIES OVERVIEW

A. INTRODUCTION

An ontology can be defined as a formal explicit description of concepts in a domain of discourse, properties of each concept describing various features and attributes of the concept, and restrictions on these properties that are specified by semantics, or rules, that follows the "rules" of the domain of knowledge. For these reasons, ontologies are useful as knowledge bases (KB) for an application attempting to add context to a particular search word or phrase. By navigating the ontologies, we can understand the context of a particular concept as well as the relationships it has with other concepts.

This chapter focuses on the syntax and construct of ontology languages. The World Wide Web Consortium (W3C) has created a language specifically for ontologies, known as the Web Ontology Language (OWL), built upon previous Web languages including the syntactic foundation of the Web, XML, and Resource Description Framework (RDF)[1]. OWL is considered an extension of RDF, and as it will be evident in the discussion below, OWL ontologies use RDF syntax to define their resources.

The importance of RDF and, in particular, OWL to building ontologies is the ability to add semantics around their concepts that is not possible with XML. XML provides meta data to describe the content of its documents. However, the description alone does not explain anything about the relationship of the content; it is void of semantics. The goal of using ontologies is to move beyond the meta data to a more semantically aware systems. RDF helps to accomplish this by providing a mechanism for creating meta data about resources on the Web, i.e., any information that can be retrieved or simply identified on the Web. RDF was developed for processing information by applications, rather than simply displaying it for humans. It creates a common framework for exchanging information across application without losing any of their meanings. However, RDF does not provide much capability for semantics, leading to the

[1] For the purpose of this thesis, a working knowledge of XML will be assumed. While certain components of XML will be defined, detailed explanation of the syntax will not be discussed.

development of OWL.[2] OWL is able to extend RDF with its semantic constructs, allowing it to define and instantiate concepts and their relationships within an ontology.

Tim Berners-Lee's vision of the Semantic Web is to move beyond the mere data presentation and meta data using HTML and XML, respectively. He argues that in order for the Semantic Web to work, systems need to have access to structured information along with a set of inference "rules" for processing automated reasoning [Berners-Lee *et al.*, 2001, 2]. Berners-Lee proposed a layered architecture for the Semantic Web as represented in Figure 1. As the figure shows, newer technologies stack on top of previous ones to achieve the realization of the Semantic Web. OWL was not part of the original stack because it was in its infant stages when Berners-Lee proposed the architecture. However, as Figure 1 shows, OWL fits naturally between the RDF and Ontology layers.

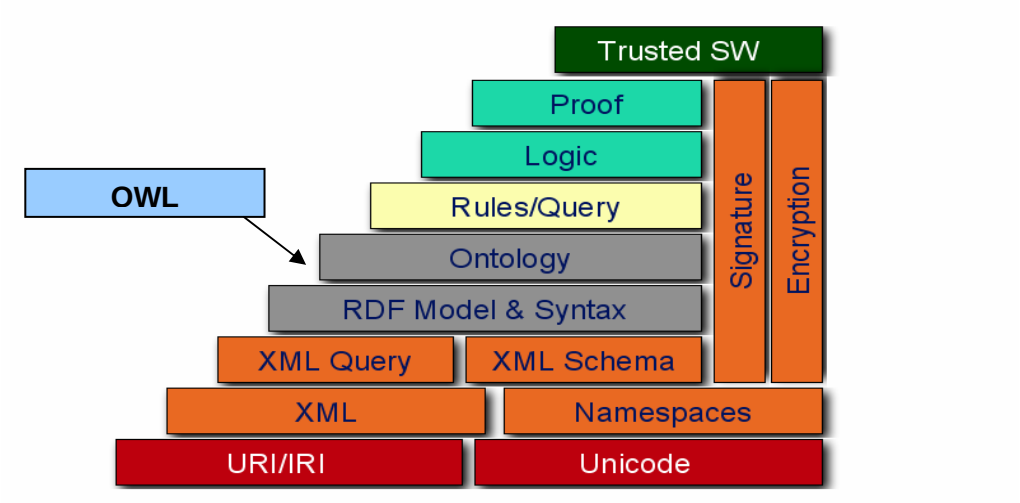

Figure 1. The Semantic Web Layers

At the bottom of the stack are the simplest forms of web identifiers, the Uniform Resource Identifier (URI) and Unicode. URI provides a means for resources to be

[2] OWL was derived from an earlier ontology language developed by DARPA called DAML+OIL. More information on DAML+OIL can be found on www.w3.org/TR/daml+oil-reference, April 2005.

uniquely identified and retrieved on the web. Unicode is an extension of ASCII that has the ability to encode the characters of all languages, rather than only the Roman alphabet.

The second layer is made up of XML and namespaces. XML is a set of syntax rules for structured documents. However, it does not enforce any semantic constraints on the documents. Namespaces are extensions of XML and they are a mechanism for differentiating elements and attributes of a particular vocabulary in order to make them globally unique. Namespaces allow different XML documents to be combined without ambiguity.

The third layer of the stack is XML query (XQuery) and XML schema. As the name implies XQuery is a query language for XML documents. XQuery was designed to query XML-based data sources as one would do to databases. XML Schema is a definition language that limits the conforming XML documents to a specific vocabulary and hierarchical structure. The fourth layer consists of RDF model and syntax. RDF is an XML-based language used to describe objects or "resources," as well as the relationship between them. This provides a data-model, known as *RDF Triple*, using simple semantics and the resources are accessed using URIs. RDF Schema is a language that describes RDF classes and properties, with semantics that specify the hierarchies of these classes and properties.

The fifth layer is the ontology itself. Ontologies are the key components to the Semantic Web because they contain the domain "knowledge" that defines concepts and their relationships to each other. Although RDF does have some capabilities to represent relationship between resources, it lacks the semantic richness and inference capacity that the OWL provides. OWL is built on RDF, adding greater vocabulary for specifying relationships between classes, individuals, property characteristics, and enumerated classes, that allows for developing rich ontologies.

The next two layers, Rules/Query and Logic provide mechanisms for querying and inferencing to provide information about the domain of interest. Although description logic of OWL lacks full expressiveness, it does have computational completeness, which is not possible with RDF.

Little is currently understood about the top two layers of the Semantic Web stack, the Proof and Trust, but they are expected to be the focus of future development efforts. The idea behind Proof and Trust is that when information is retrieved from the Semantic Web, it must be proven and trusted as the right answer. For instance, if one source states that the country of Armenia is in the continent of Europe and another source states that it is in the continent of Asia, which source should be trusted? These inconsistencies would make it impossible for the success of the Semantic Web. However, as the focus shifts to proof checking mechanisms and digital signatures, these layers will provide a step closer to Berners-Lee's goal for the next generation of the Web [Palmer, 2001, 11].

The remainder of this chapter addresses the syntax and constructs of RDF and OWL. It is organized as follows. Section B is a brief overview of RDF while section C discusses the syntax and constructs of OWL.

B. RDF PRIMER

RDF was developed to represent information about Web resources. The term "resource," which has a broad meaning, can be simply thought of as the electronic file available out on the Web [Daconta et al., 2003, 12]. Rather than just displaying information for human consumption, the RDF was developed to allow machines and applications to process information. Such an exchange of information is possible with the RDF common framework, which uses parsers and data processing mechanisms. The RDF capabilities to uniquely identify resources and share information across all platforms lay the core foundation for semantically richer languages, such as OWL.

1. RDF Triple

There are three necessary RDF components for identifying a piece of information. Known as the RDF *triple*, it consists of subject, predicate, and object. Consider a common English statement below.

http://www.ontology.net/geography.html has a creator whose value is Ann Lee.

The above statement is broken down into the RDF triple elements as follows.

1. SUBJECT: http://www.ontology.net/geography.html
2. PREDICATE: **creator**
3. OBJECT: **Ann Lee**

In order for machines to understand the meaning or "knowledge" of this statement, the components of the English sentence must be formatted in such a way that they are machine-consumable. RDF accomplishes this by using what the URI references (URIrefs). URIrefs are RDF's primary mechanism for specifying subject, predicate and object in its statements. An URIref has two parts, namely an URI joined with a fragment identifier. For example, http://www.ontology.net/geography.html#Country is a combination of the URI http://www.ontology.net/geography.html and the fragment identifier *Country* separated by the # sign. Unlike Uniform Resource Locators (URLs), URIrefs do not require direct connection to an actual Web resource.

The RDF triples are often depicted as a graph using nodes and arcs. For example, the English statement above is represented by the graph shown in Figure 2.

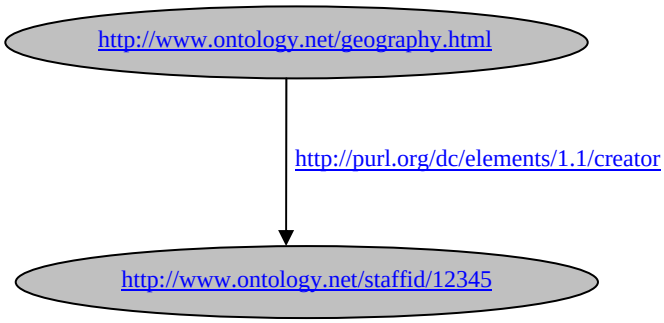

Figure 2. RDF Triple Model

As the figure shows, the subject and object are represented by nodes, and the predicate by an arc from the subject to the object node. Also, the predicate and object are specified by an URIref, rather than simple values of "creator" and "Ann Lee", as in the English statement above. The URIrefs uniquely associates any property or value to a particular resource identifier. That is, the usage of the property "creator" may have different meanings for different developers or applications. In order to clarify the exact meaning of "creator" as implied by the developer, the term is associated with an URIref to make the description unambiguous to the user. Therefore, depicting the predicate as

"http://purl.org/dc/elements/1.1/creator" helps distinguish it from other meanings of "creator," such as the one with resource of "http://www.anotheruser.org/term/creator." Furthermore, the use of URIref to identify the property makes it possible to augment additional information. For example, the object URIref of a particular RDF statement may be used as a subject of another RDF statement.

It is important to understand that one resource may be part of several RDF statements. Consider the diagram below.

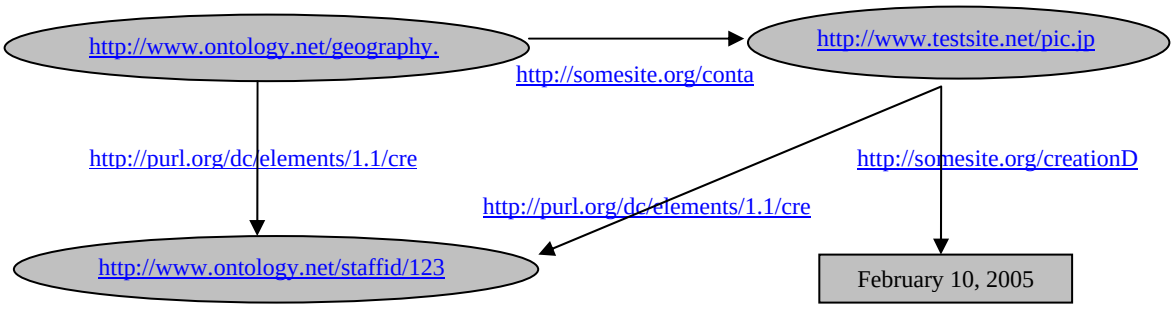

Figure 3. Multiple RDF Statements Interconnected

Figure 3 illustrates how multiple statements interconnect with one another, and provide multiple layers of information for a given node. Each arc corresponds with a RDF triple. Thus, Figure 3 represents four separate RDF triple statements. Also, even though literals may not be used as subjects or predicates, objects may take on a constant value. The node http://www.testsite.net/pic.jpeg, representing a picture file, has for its creation date the literal value of "February 10, 2005."

Like XML, RDF uses namespaces to abbreviate for URIs. For example, the prefix `myont` is the namespace representation for the URI http://www.ontology.net/geography#. Future references to this resource, then, may be written as `myont:`, followed by the fragment identifier such as `myont:Country`. Some common namespace prefixes are as listed in Table 1.

Prefix	URI
`rdf:`	http://www.w3.org/1999/02/22-rdf-syntax-ns#
`rdfs:`	http://www.w3.org/2000/01/rdf-schema#
`dc:`	http://purl.org/dc/elements/1.1/
`owl:`	http://www.w3.org/2002/07/owl#
`xsd:`	http://www.w3.org/2001/XMLSchema#

Table 1. List of Common Namespaces

Within the RDF namespace, there are defined constructs that denote different relationship semantics. For instance, the property `rdf:type` is used to define the various kinds of relationships that exist between resources. This is especially useful with RDF Schema specification vocabularies of `Class` and `subClassOf`, where `rdf:type` is analogous to the `instanceOf` property in object-oriented languages. RDF constructs, such as `rdf:type`, will be discussed further in the sections below.

2. RDF Schema

RDF Schema, the semantic extension of RDF, allows for the development of an application-neutral vocabulary for defining class and subclass hierarchies, as well as properties to describe these classes. Unlike object-oriented languages that RDF Schema is often compared to, the RDF vocabulary defines properties in terms of resource classes. That is, one can create new properties out of existing ones simply by adding to the original property specifications without redefining the original description and restrictions of the class. This is the benefit of RDF Schema's property-centricity, having the ability to extend the existing resource descriptions.

The facilities of RDF Schema are predefined with its own set of RDF vocabulary under the resource, http://www.w3.org/2000/01/rdf-schema#. The vocabulary may be referenced using the `rdfs` namespace as specified above. The RDF examples below will use the `rdfs` and `rdf` (URI: http://www.w3.org/1999/02/22-rdf-syntax-ns#) namespaces.

3. Classes

RDF Schema categorizes similar “kinds of things” into *classes.* Syntactically, a class in RDF Schema is any resource that has the `rdf:type` property value of `rdfs:class`. The members of a class, known as *instances*, are also known as class

extensions. Multiple classes may have the same set of instance, which have all sets of properties from each class. Furthermore, a class may be defined by a set of its own extensions and/or extensions of other classes.

A *Subclass* is a child class, which inherits all the characteristics from its parent classes. If class B is a subclass of class A, then all the instances of B are also instances of A. The superclass-subclass relationship is called an "is-a" relationship, meaning an instance of class B is also an instance of class A. If class C is also a subclass of class A, then class C is a sibling class to class B. They share all the same traits inherited from class A, along with their unique properties that differentiate them. The syntax `rdfs:subClassOf` is used to define a subclass. Table 2 lists class relevant RDF constructs.

RDF Construct	***Description***
`rdf:type`	Specifies that a given resource is an instance of some class.
`rdfs:subClassOf`	Specifies that all the instances of one class are also instances of another class.
`rdfs:label`	Used to provide human-readable names for the resources.
`rdfs:comment`	Used to provide human-readable descriptions of the resources.

Table 2. RDF Class Constructs

4 Properties

RDF property is the *predicate* relationship between the subject and object resources. All RDF properties have the `rdf:type` value of `rdfs:Property`. Like classes, properties are arranged in hierarchies where `rdfs:subPropertyOf` construct denotes a taxonomic, superclass-subclass relationship. Thus, if the property Y is a subproperty of X, then all resources related to property Y are also related to property X. There are three RDF Schema syntax used to define properties. The range of a property, specified by `rdfs:range`, indicates the property's allowed set of values. The property's domain, `rdfs:domain`, is used to show that the property is applied to a designated class or set of classes. These property semantics, further defined in Table 3, are used to describe RDF properties.

RDF Construct	*Description*
`rdfs:subPropertyOf`	Specifies that all the properties are also subproperties of another property.
`rdfs:range`	Specifies the class instance or a literal that a given property must take one as its value.
`rdfs:domain`	Specifies the class that "owns" the property. That is, it associates the property with the class it modifies and asserts that the subjects of such property statements must belong to the instance of the class.

Table 3. RDF Property Constructs

5. Other RDF Vocabularies

RDF containers are predefined syntax used to represent collections of resources. There are three container vocabularies in RDF Schema, namely `rdf:Bag`, `rdf:Seq`, and `rdf:Alt`, which are specified under the class `rdfs:Container`. The *bag* (`rdf:Bag`) defines a group of resources or literals where the order of its members is not significant. The *sequence* (`rdf:Seq`) is a group resources or literals where the order of them are, in fact, relevant, whether it is alphabetical, numeric, or other types of ordering. The *alternative* (`rdf:Alt`) is a group of resources or literals that are "alternatives" to the other containers. That is, all the members are alternates of one another. All the containers described above allow duplicate members in its list.

RDF collection differs from the RDF containers in that it is a *closed* list. In other words, it is an exhaustive list of members, exclusive to other possible candidates. In specifying the RDF collection, the syntax `rdf:List` describes the list while `rdf:first` property refers to the first-member relationship and `rdf:rest` property refers to the rest-of-list relationship.

Putting all the RDF Schema vocabularies together, here is an example of an RDF document in Figure 4.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#
         xmlns:rdfs=http://www/w3/org/2000/01/rdf-schema#
         xml:base=http://www/geodesy.org/water/naturally-occurring>

<rdfs:Class rdf:ID="River">
   <rdfs:subClassOf rdf:resource="#Stream"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Stream">
   <rdfs:subClassOf rdf:resource="#NaturallyOccurringWaterSource"/>
</rdfs:Class>

<rdf:Property rdf:ID="emptiesInto">
   <rdfs:domain rdf:resource="#River"/>
   <rdfs:range rdf:resource="#BodyOfWater"/>
</rdfs:Property>

<rdf:Property rdf:ID="hasLength">
   <rdfs:domain rdf:resource="#River"/>
   <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
    schema#Literal"/>
</rdfs:Property>
…
</rdf:RDF>
```

The RDF statement in Figure 4 defines both classes and properties. The classes are `River` and `Stream` identified by the `rdf:ID` syntax, where `River` is a subclass of `Stream` and `Stream` is a subclass of `NatuallyOccurringWaterSource`, which is defined elsewhere in the RDF document. Also, there are two properties, `emptiesInto` and `hasLength`, specified by `rdf:ID`. While both properties have a domain value of class `River`, the range of `emptiesInto` is the `BodyOfWater` class and the range of `hasLength` is a literal string. Thus, instances of `River` having these properties must select an instance of `BodyOfWater` as the value for `emptiesInto` property and a literal string for the value of `hasLength` property.

Although RDF provides certain semantics for knowledge representation for systems, it still lacks the semantic richness for creating meaningful ontologies and capability for inferences. To this end OWL was developed to address the semantic limitations of RDF.

C. WEB ONTOLOGY LANGUAGE (OWL): BUILDING AN ONTOLOGY

OWL builds on RDF and RDF Schema, with enhanced capability to describe classes and properties. Like RDF, OWL uses URIs and description framework for a wide distribution across systems, necessary scalability for the web, compatible Web standards, extensibility and openness. OWL's rich semantics provide better interpretability than XML, RDF, and RDF Schema, making it ideal as a platform independent, machine-process language. OWL is the most appropriate language available to express explicitly the meaning of terms in a domain as well as relationships between them.

There are three OWL sublanguages, namely OWL Lite, OWL DL, and OWL Full, in increasingly expressive sequence.

OWL Lite, the least expressive of the sublanguage, is used primarily to support classification hierarchy and simple restriction features. And, while OWL Lite allows cardinality restrictions, it is limited to the values of 0 and 1. This sublanguage is an ideal choice for developing quick and simple taxonomies.

OWL-DL supports maximum expressiveness while maintaining the reasoning system's computational completeness and decidability. In other words, all inferences are guaranteed for computation and these computations will be completed in a finite amount of time. With some exceptions, such as *type separation*[3], OWL-DL includes all constructs of the OWL language and it is derived from its conformity to *description logic*. OWL-DL was specifically designed to allow logic inferencing and has ideal reasoning system computational properties.

OWL Full allows maximum expressiveness and full capability of the RDF syntax without the computational guarantees. An OWL Full class may be treated as an individual and a collection of individuals simultaneously. Also, OWL Full permits an ontology to add meanings of pre-defined (RDF or OWL) vocabularies. However, most reasoning application do not support all the features of OWL Full in order to maintain its computation decidability and completeness.

[3] This is when class cannot take on the role of an individual or property; or, property cannot take on the role of a class or individual.

When choosing the most appropriate OWL ontology sublanguage, developers should evaluate the usage of the ontology. The need for expressiveness and computational simplicity usually determines the choice between OWL Lite and OWL DL. Meanwhile, the choice between OWL DL and OWL Full depends on the need for RDF Schema's meta-modeling facilities. However, the developer should understand the trade-off between greater flexibility and reasoning capability when choosing the right sublanguage of OWL.

For the purposes of this thesis and the Ontology Aided Knowledge Discovery Assistant (OAKDA) application developed in this thesis, OWL-DL will be the designated sublanguage. Its computational completeness and decidability are crucial to the ontology development and use for the OAKDA application. However, most of the OWL class syntax and constructs discussed below applies to all sublanguages of OWL. It is in the description of OWL properties, restrictions, and complex classes where the focus will shift to OWL-DL. Unless specified otherwise, reference to OWL should be assumed to mean OWL-DL.

Throughout this chapter and the next, many examples will be used to illustrate the syntax, constructs and methodology for building ontologies. The goal at the end of these two chapters is to develop a sample ontology that will be used in the OAKDA application. For the purpose of this thesis, the authors chose geography as the ontology domain of context. The main motivation is its familiarity and a relatively high general interest in the topic by many readers.

Numerous GUI based ontology editors exist for developing ontologies. These editors simplify ontology development by generating the OWL code from the graphical specification. In particular, Protégé[4] is an ontology editor that ontology developers can use without the knowledge of the syntax and constructs of OWL. Although Protégé does hide the details of OWL, it is crucial for all OWL-DL ontology developers to understand the constructs and semantics of OWL. Protégé will be used and referenced throughout this thesis to illustrate and visually display examples of the Geography OWL ontology.

[4] Protégé is an open-source application developed at Stanford University. More information is available at http://protege.stanford.edu/index.html, May 2004.

1. Namespaces and Ontology Headers

In developing an OWL ontology, it is typical to begin the document by stating the set of vocabularies that will be used throughout the ontology. This is done by declaring namespaces. Figure 5 shows namespace declarations for the Geography ontology.

```
<rdf:RDF
    xmlns="http://a.com/ontology/Geography#"
    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">
```

Figure 5. Sample Namespace Declaration

The first is the default namespace, referring to the Geography ontology itself. The next three namespaces are W3C's predefined vocabularies. While the OWL constructs are defined under http://www.w3.org/2002/07/owl, and since OWL is built on RDF, RDF Schema, and XML Schema, all three URLs are listed as necessary namespaces for building an OWL ontology.

Once the namespaces are listed, a set of assertions for the ontology can be grouped under the tag, `owl:Ontology`. This is the place to list the meta-data information about the ontology. Table 4 shows meta data constructs commonly used under the `owl:Ontology` heading.

Construct	*Description*
`rdf:about`	Specifies a name or reference of the ontology. If the value of "", then the default value is the base URI which contains the ontology.
`rdfs:comment`	Provides a place for comments and annotations regarding the ontology.
`owl:priorVersion`	Provides a capability for a version control system.
`owl:imports`	Allows importing of other ontologies, with all of their assertions, in the current ontology. This should be used in coordination with the namespace, which allows the references to the imported ontologies.
`owl:AnnotationProperty`	Provides to declare properties that are used as annotations.
`rdfs:label`	Allows natural language labels for the ontology.

Table 4. Meta Data Constructs

Example of these meta data constructs is demonstrated in the OWL statement in Figure 6.

```
<owl:Ontology rdf:about="">
  <rdfs:comment>An example OWL ontology</rdfs:comment>
  <owl:priorVersion>
    <owl:Ontology rdf:about="
     http://a.com/ontology/02102004/Geography"/>
  </owl:priorVersion>
  <owl:imports
   rdf:resource="http://www/geodesy.org/water/naturally-
   occurring"/>
  <rdfs:comment>Developed as part of the OWL Ontology Development
   Tutorial by Ann Lee and Edward Powers.
  </rdfs:comment>
  <rdfs:label>Geography Ontology</rdfs:label>
</owl:Ontology>
```

Figure 6. Example OWL Meta Data

2. Basic Components of OWL

The basic components of OWL are classes, properties, individuals, and the relationship between the individuals. These components are discussed in detail in the following sections.

In OWL, classes may be defined in a variety of ways. Class description, as termed by the W3C, allows six methods for describing a class:

1. A class identifier (URI reference)
2. A complete list of individuals that combined to form instances of a class (enumeration)
3. A property restriction
4. An intersection of two or more class descriptions
5. An union of two or more class descriptions
6. A complement of a class description

The six types of class definition will be discussed in detail in the sections below.

3. Defining OWL Classes

Similar to RDF, the most basic component of OWL is the class. In creating an ontology, classes are the mechanism for abstracting groups of resources that share similar characteristics. An OWL class is associated with a set of individuals, known as the *class extension*, which are instances of the class. It is crucial to distinguish between a class and its class extension. Although they are related to each other, they are not equal. Thus, two different classes can have the same instances without conflict. Those instances will have characteristics of both classes.

The most straightforward method of defining a class is by specifying a name, represented syntactically using an URI. All OWL classes belong to a superclass `owl:Thing`. In other words, all the user-defined classes are subclasses of `owl:Thing`.

For example, in the `Geography` ontology, the classes `Ocean`, `Mountain`, and `Country` are defined in OWL as stated in Figure 7.

```
<owl:Class rdf:ID="Ocean"/>
      <owl:Class rdf:ID="Mountain"/>
      <owl:Class rdf:ID="Country"/>
```

Figure 7. OWL Class Definition by Name

Here, the classes `Ocean`, `Mountain`, and `Country` are the named classes. Class definitions are specified with `owl:Class` and the `rdf:ID` identifies the name. `owl:Class` construct is a subclass of `rdfs:Class`, which an additional description logic component[5]. Within the ontology document, references to these classes are stated

[5] In OWL Full, these two statements, `owl:Class` and `rdfs:Class`, are equivalent.

as #Ocean, #Mountain, and #Country. OWL uses the RDF Schema syntax rdf:ID to introduce the class name as part of the class definition.

The basic component of building a taxonomy of classes is the rdfs:subClassOf construct. As in a tree structure, this syntax relates the specific subclass to the general superclass. Thus, all the instances of the subclass are instances of the superclass. This relationship is transitive so that if class B is a subclass of class A and class C is a subclass of class B, then class C is a subclass of A. The OWL statements in Figure 8 exemplify the subclass relationship.

```
<owl:Class rdf:ID="Mountain"/>
   <rdfs:subClassOf rdf:resource="#BodyOfLand"/>
   …
</owl:Class>

<owl:Class rdf:ID="Volcano"/>
   <rdfs:subClassOf rdf:resource="#Mountain"/>
   …
</owl:Class>
```

Figure 8. Basic Subclass Specification

The examples in Figure 8 define two OWL classes, Mountain and Volcano. It explicitly states that class Mountain is a subclass of BodyOfLand and class Volcano is a subclass of Mountain. Therefore, by rules of subsumption, Volcano is also a subclass of BodyOfLand, inheriting all the characteristics of BodyOfLand as well as additional properties of Mountain.

There are two components to a class definition. The first part is the name declaration, or reference, and the second part is the list of class description or restriction. Furthermore, subclasses inherit the properties and their restriction from their parent classes. And every restriction specified as part of the class definition further confines the instances of that class. In other words, the individuals are the instances of the intersection of all the restrictions of the class. Thus, in the Geography example, Volcano is bound by all the properties of the Mountain and BodyOfLand classes in addition to its own set of restrictions.

a. Disjoint Classes (`disjointWith`)

When classes are disjointed from one another, they cannot share the same individuals. In other words, if the classes `Ocean` and `Lake` are disjointed from each other, they cannot have an individual that is a member of both classes.

```
<owl:Class rdf:ID="LandlockedCountry">
  <rdfs:subClassOf rdf:resource="Country">
  <owl:disjointWith rdf:resource="CoastalCountry"/>
  <owl:disjointWith rdf:resource="IslandCountry"/>
  <owl:disjointWith rdf:resource="PeninsulaCountry"/>
  <owl:disjointWith rdf:resource="ArchipelagoCountry"/>
</owl:Class>
```

Figure 9. Disjoint Classes

The OWL statement in Figure 9 specifies that the `LandlockedCountry` class is disjoint from all the classes listed in the statement. However, this does not assert that the listed classes are disjointed from each other. That is, by this description, `CoastalCountry` is not disjointed with `IslandCountry`. In order to state mutual disjointed relationships, every class must be asserted with the `owl:disjointWith` relationship.

4. Individuals

As briefly discussed, the members or instances of classes are referred to as *individuals*. There are two ways of defining an individual in OWL. The two statements are identical in meaning.

```
<Ocean rdf:ID="PacificOcean"/>

            Or
<owl:Thing rdf:ID="PacificOcean"/>

<owl:Thing rdf:about="#PacificOcean"/>
   <rdf:type rdf:resource="#Ocean"/>
</owl:Thing>
```

Figure 10. Instantiating OWL Individuals

The first method of declaring an individual is by simple instantiating the class as in the first statement in Figure 10. The second method uses `rdf:type`, like in RDF, to link an individual to its class and it is a two-part statement. It should be clarified that

these two statements do not need to be adjacent. In fact, they do not even need to be part of the same ontology. Since Web ontologies are designed for distribution, they can be augmented or imported within other ontologies. Thus, the instantiation and use of individuals can occur in two separate ontology documents.

5. Properties

Properties assert general information about a class and specify concrete information about individuals of that class. The sections below discuss the different types of properties and how they are used as class restrictions.

a. Defining Properties

There are two types of properties, namely *datatype* and *object* properties. Datatype properties specify the relationships between individuals and RDF literals or XML Schema datatypes. Object properties represent the relationships between individuals of one or more classes. Several methods can be used to create a property relationship, whether it is datatype or object property. The most common method to specify an object property is limiting the domain and range of the property to individuals of certain classes. In Figure 11, the `hasBoundary` property has a range restriction of all the individuals of the `BodyOfLand` class and a domain restriction of all the individuals of `BodyOfWater` class.

```
<owl:ObjectProperty rdf:ID="hasBoundary">
    <rdfs:range rdf:resource="#BodyOfLand"/>
    <rdfs:domain rdf:resource="#BodyOfWater"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasBorder">
    <rdfs:domain rdf:resource="#BodyOfLand"/>
</owl:ObjectProperty>
```

Figure 11. Property Restriction Using Domain and Range

Unlike the `hasBounday` property, the `hasBorder` property in Figure 11 shows only a domain restriction. By not explicitly stating the range restriction within the property definition, it is implied that all individuals, regardless of class will have the the default range `owl:Thing`.

Similar to classes, properties may be organized as a hierarchy. Likewise, a property may be defined as a subproperty, or a *specialization*, of another property. Figure 12 shows examples of the property subsumption using the construct `rdfs:subPropertyOf`.

```
 <owl:ObjectProperty rdf:ID="hasCountryDescriptor">
    <rdfs:domain rdf:resource="#Country"/>
    <rdfs:range rdf:resource="#CountryDescriptor"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="hasOfficialLanguage">
    <rdfs:range rdf:resource="#Language"/>
     <rdfs:subPropertyOf rdf:resource="#hasCountryDescriptor"/>
</owl:ObjectProperty>
```

Figure 12. Property Subsumption Examples

Using `rdfs:subPropertyOf`, the `hasOfficialLanguage` object property is a type of `hasCountryDescriptor` property, inheriting all of the parent property's characteristics. The definition of `hasCountryDescriptor` is defined by the domain and range restrictions. Although the `hasOfficialLanguage` property inherits these restrictions from the parent property, by explicitly assigning a new range, the `Language` class, it further restricts the possible individuals that can fill the value of this property. Therefore, the range of this property is not merely the instances of the `CountryDescriptor` class, as specified in the parent property, it is further confined to the instances of the `Language` class.

Using property restrictions, it is now possible to expand on the simple class definition. The class `CoastalCountry`, for example, includes a property restriction as part of its definition.

```
<owl:Class rdf:ID="CoastalCountry">
  <rdfs:subClassOf rdf:resource="#Country"/>
  <rdfs:subClassOf>
     <owl:Restriction>
        <owl:onProperty rdf:resource="#hasCoastline"/>
        <owl:minCardinality rdf:datatype="&xsd;nonNegativeInteger">1
        </owl:minCardinality>
     </owl:Restriction>
  </rdfs:subClassOf>
  …
</owl:Class>
```

Figure 13. Property Restriction in Class Description

Consider the restriction listed under the second `rdfs:subClassOf` syntax in Figure 13. This subclass declaration is of an unnamed class, or an *anonymous* class, used as part of the `CoastalCountry` class definition. Under the anonymous class description, the restriction uses the `owl:minCardinality` to state that the individuals of this class must have at least one associated `hasCoastline` property value. Thus, the complete definition of `CoastalCountry` states that all individuals of this class are an instance of the `Country` class and meet the minimum cardinality restriction of the `hasCoastline` property. More will be discussed about the cardinality property in Section 7.b. of this chapter.

b. Properties and Datatypes

Datatype property values range between RDF literals (`rdfs:Literal`) and XML Schema datatypes. Table 5 is the list of XML Schema types used with OWL datatype properties.

xsd:string	xsd:normalizedString	xsd:boolean
xsd:decimal	xsd:float	xsd:double
xsd:integer	xsd:nonNegativeInteger	xsd:postiveInteger
xsd:nonPositiveInteger	xsd:negativeInteger	xsd:unsignedByte
xsd:long	xsd:int	xsd:short
xsd:unsignedLong	xsd:unsignedInt	xsd:unsignedShort
xsd:hexBinary	xsd:base64Binary	
xsd:dateTime	xsd:time	xsd:date
xsd:gYear	xsd:gMonthDay	xsd:gDay
xsd:byte	xsd:gYearMonth	xsd:gMonth
xsd:anyURI	xsd:token	xsd:language
xsd:NMTOKEN	xsd:Name	xsd:NCName

Table 5. XML Schema Datatypes

6. Property Characteristics

In addition to domains and ranges, other OWL constructs allow greater semantic expressiveness. These OWL property characteristics provide the means for classification and reasoning on the ontology.

a. Transitive and Symmetric Properties

OWL properties can take on transitive attributes. If a property is defined as transitive to another property, the values of the property may have inferred relationships with one another. Mathematically, a transitive property is expressed as:

```
P(a,b) and P(b,c) implies P(a,c)
```

In the Geography ontology, `locatedIn` is a transitive property. Figure 14 shows how transitive property is defined in OWL.

```
<owl:ObjectProperty rdf:ID="locatedIn">
  <rdfs:type rdf:resource="&owl;TransitiveProperty"/>
  <rdfs:domain rdf:resource="&owl;Thing"/>
  <rdfs:range rdf:resource="#PoliticalGeography"/>
</owl:ObjectProperty>

<Region rdf:ID="VaticanCity"/>
  <locatedIn rdf:resource="#Rome"/>
</Country>

<Region rdf:ID="Rome"/>
  <locatedIn rdf:resource="#Italy"/>
</Country>
```

Figure 14. Transitive Property Defined

The OWL statements in Figure 14 explicitly state that the individual `VaticanCity` is located in individual `Rome` and that individual `Rome` is located in individual `Italy`. Since `locatedIn` is defined as a transitive property, it is inferred that `VaticanCity` is located in the region of `Italy`.

Another OWL property construct is the symmetric attribute. If a property is designated as symmetric, then for any values a and b, there is the following relationship:

```
P(a,b), iff P(b,a)
```

In the Geography ontology, `adjacentTo` is a symmetric property. Figure 15 shows how symmetric property is defined in OWL.

```
<owl:ObjectProperty rdf:ID="adjacentCountry">
  <rdfs:type rdf:resource="&owl;SymmetricProperty"/>
  <rdfs:domain rdf:resource="#Country"/>
  <rdfs:range rdf:resource="#Country"/>
</owl:ObjectProperty>

<Country rdf:ID="Nigeria"/>
  <locatedIn rdf:resource="#Africa"/>
  <adjacentCountry rdf:resource="#Cameroon"/>
</Country>
```

Figure 15. Symmetric Property Defined

The OWL statements in Figure 15 explicitly state that the individual `Nigeria` has the `adjacentCountry` property value of the individual `Cameroon`. However, because `adjacentCountry` is a symmetric property, it can is inferred that `Cameroon` has the `adjacentCountry` property value of `Nigeria`. In order for a property to be symmetric, it must have the same domain and range value. It is invalid to have a symmetrical relationship between two individuals that belong to different classes.

b. Functional Property

An OWL functional property states that there is only one value associated with the property. If an individual designates more than one value of a functional property attribute, then it is assumed that those values are the same. Mathematically, a functional property is stated as follows:

P(a,b) and P(a,c) implies a = c

In the Geography ontology, `hasCapitalCity` is a functional property. Figure 16 shows how a functional property is defined in OWL.

```
<owl:ObjectProperty rdf:ID="hasCapitalCity">
  <rdfs:type rdf:resource="&owl;FunctionalProperty"/>
  <rdfs:domain rdf:resource="#Country"/>
  <rdfs:range rdf:resource="#CapitalCity"/>
</owl:ObjectProperty>
```

Figure 16. Functional Property Defined

For every individual of the `Country` class, there can only be one value associated with the `hasCapitalCity` property. Thus, for the country individual `UnitedStates`, if its `hasCapitalCity` property is filled with `WashingtonDC` *and* `DistrictOfColumbia`, it can be assumed that these two values are the identical.

c. InverseOf Property

If a property is defined with the `owl:inverseOf` construct of another property, then they have the following relationship:

$P_1(a,b)$ iff $P_2(b,a)$

The domain and range determines the direction of the property. That is, the property states the relationship from the domain, the subject, to the range, the object. However, it is often necessary to define another property that states the relationship in the opposite direction. Therefore, the “inverse of” property inverses the domain and range of the initial property so that the domain becomes the new range and the range becomes the new domain. Figure 17 shows how two properties have an "inverse of" relationship using the `owl:inverseOf` construct.

```
<owl:ObjectProperty rdf:ID="hasCapitalCity">
  <rdfs:type rdf:resource="&owl;FunctionalProperty"/>
  <rdfs:domain rdf:resource="#Country"/>
  <rdfs:range rdf:resource="#CapitalCity"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="belongsToCountry">
  <owl:inverseOf rdf:resource="#hasCapitalCity "/>
  <rdfs:domain rdf:resource="#CapitalCity "/>
  <rdfs:range rdf:resource="#Country"/>
</owl:ObjectProperty>
```

Figure 17. InverseOf Property Defined

The OWL statements in Figure 17 indicated that the property belongsToCountry is an inverse property of hasCapitalCity. Hence, if the individual Madrid has a belongsToCountry relationship with individual Spain, then by the rules of the "inverse of" property, Spain will automatically have a hasCapitalCity relationship with Madrid. Also notice the domain and range of the owl:inverseOf properties. The domain of one property is the range of another, and vise versa.

d. Inverse Functional Property

The inverse functional property combines the traits of the “inverse of” and functional properties. It indicates that it has an inverse of relationship with another property, which must be a functional property.

The properties `hasCapitalCity` and `belongsToCountry`, shown in Figure 16, are in fact inverse functional properties. Since hasCapitalCity is a functional property, its inverse must be an inverse *functional* property. Mathematically, an inverse functional property is expressed as follows:

P(a,b) and P(c,b) implies a=c

In the Geography ontology, the `belongsToCountry` property is defined as inverse functional property, shown in Figure 18.

```
<owl:ObjectProperty rdf:ID="belongsToCountry">
  <rdfs:type rdf:resource="&owl;InverseFunctionalProperty"/>
  <owl:inverseOf rdf:resource="#hasCapitalCity "/>
  <rdfs:domain rdf:resource="#CapitalCity "/>
  <rdfs:range rdf:resource="#Country"/>
</owl:ObjectProperty>
```

Figure 18. Inverse Functional Property Defined

If the individual `WashingtonDC` has a `belongsToCountry` relationship with `UnitedStated`, *and* `DistrictOfColumbia` also has a `belongsToCountry` relationship with `UnitedStated`, then by the rules of inverse functional properties, `WashingtonDC` and `DistrictOfColumbia` are identical objects.

An inverse functional property is similar to a unique key in a database. The domain and range of the inverse functional properties create a unique identifier combination, where one element of the domain is always associated with a particular domain of the range.

7. Property Restrictions

In addition to the variety of property characteristics discussed above, classes may use *property restrictions* as part of their description. The restrictions are defined by the syntax `owl:Restrictions` and `owl:onProperty`.

a. someValuesFrom and allValuesFrom Restrictions

Although the domain and range restrictions of a property apply to all classes using that property, a class definition may further confine the property value at the *local* level. The restriction syntax `owl:someValuesFrom` states that for every instance of the class using that particular property, the values of the property must have *at least one* individual of the class specified by the `owl:someValuesFrom` clause.

Figure 19 shows a class definition of `IslandCountry` and the use of `owl:someValuesFrom` to limit the `hasLandType` property value.

```
<owl:Class rdf:ID="IslandCountry">
  <rdfs:subClassOf rdf:resource="#Country"/>
  …
  <rdfs:subClassOf>
     <owl:Restriction>
        <owl:onProperty rdf:resource="#hasLandType"/>
        <owl:someValuesFrom rdf:resource="#Island"/>
     </owl:Restriction>
  </rdfs:subClassOf>
  …
</owl:Class>
```

Figure 19. owl:someValuesFrom Example

The `owl:someValuesFrom` restriction states that the individuals of `IslandCountry` with the property `hasLandType` value must have at least one individual belonging to the `Island` class. This restriction on the property only applies to this class and its subclasses and not other that use the `hasLandType` property, such as its sibling class `LandlockedCountry`.

The `owl:allValuesFrom` restriction states that *if* an instance of a class has any value for this restricted property, they all must be a value from the specified class of individuals. Unlike the `owl:someValuesFrom` restriction, which requires the property to have at least one value, the `owl:allValuesFrom` restriction allows the property to have a null value. The `owl:allValuesFrom` is often used in conjunction with `owl:someValuesFrom` as a closure axiom for a property restriction. If the developer's intention is to restrict the property value to only individuals a certain class, rather than at least one value, then `owl:allValuesFrom` should be used with `owl:someValuesFrom`.

```
<owl:Class rdf:ID="IslandCountry">
  <rdfs:subClassOf rdf:resource="#Country"/>
  <rdfs:subClassOf>
     <owl:Restriction>
        <owl:onProperty rdf:resource="#hasLandType"/>
        <owl:someValuesFrom rdf:resource="#Island"/>
        <owl:allValuesFrom rdf:resource="#Island"/>
     </owl:Restriction>
  </rdfs:subClassOf>
  …
</owl:Class>
```

Figure 20. owl:someValuesFrom & owl:allValuesFrom Example

Figure 20 shows a class description using both `owl:someValuesFrom` and `owl:allValuesFrom` restrictions. This implies that all individuals of `IslandCountry` with the `hasLandType` property value can *only* have values of `Island` individuals.

b. Cardinality Restriction

Cardinality specifies the minimum, maximum, or the exact number of values in a property relationship. Unless cardinality is specified, it is assumed that there is an unlimited possible property values. Cardinality is important when the class description is based on a specific number of property attributes. For example, when defining a bi-coastal area, the class description must specify that it borders the ocean on two sides of the land. Likewise minimum and maximum cardinalities put a specific restriction on a class.

Figure 21 shows a Geography class `BicoastalCountry` definition.

```
<owl:Class rdf:ID="BicoastalCountry">
  <rdfs:subClassOf rdf:resource="#CoastalCountry"/>
  <rdfs:subClassOf>
     <owl:Restriction>
        <owl:onProperty rdf:resource="#bordersOcean"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">2
        </owl:cardinality>
     </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Figure 21. Cardinality Example

Based on the OWL statement, the `owl:cardinality` constructs restrict the class to have two `bordersOcean` property values. Therefore, all individuals of `BicoastalCountry` using this property must specify exactly two property values.

Likewise, `owl:maxCardinality` and `owl:minCardinality` set the upper and lower bounds of the property cardinality. Used in combination, they limit the property to a numeric range.

c. owl:hasValue Restriction

The `owl:hasValue` construct restricts the class definition by specifying the exact value of the specified property. Hence, individuals of the class must have at least one of its property values equal to the `owl:hasValue` restriction.

```
<owl:Class rdf:ID="Lake">
  <rdfs:subClassOf rdf:resource="#BodyOfWater"/>
  <rdfs:subClassOf>
     <owl:Restriction>
        <owl:onProperty rdf:resource="#hasSaline"/>
        <owl:hasValue>
           <SaltContent rdf:ID="NotSalty"/>
        </owl:hasValue>
     </owl:Restriction>
  </rdfs:subClassOf>
  …
</owl:Class>
```

Figure 22. owl:hasValue Example

Figure 22 is a class definition of `Lake`, which specifies that its `hasSaline` must have a property value of `NotSalty`. This declares that all individuals of `Lake` must have at least one `hasSaline` property value that equals `NotSalty` to satisfy this class requirement. Similar to the `allValuesFrom` and `someValuesFrom`, this restriction is only applied to the local class.

d. Equivalent Classes and Properties

The `owl:equivalentClass` indicates that two classes have exactly the same class extensions or set of individuals. This construct has a variety of use. First, when adopting multiple ontologies, it is used to map one class to another if they are

identical with different class names. Second, it provides another method for defining classes. The `owl:equivalentClass` allows classes to be defined by a set of restrictions.

```
<owl:Class rdf:ID="BodyOfWater">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#consistsOf"/>
      <owl:someValuesFrom rdf:resource="#Water"/>
      <owl:allValuesFrom rdf:resource="#Water"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Figure 23. owl:equivalentClass Example

The OWL statements of Figure 23 indicate that the definition of `BodyOfWater` class is exactly equivalent to all the classes that meet the restriction requirement that the `consistsOf` property value is only individuals of the `Water` class. Although this description may be used with the `rdfs:subClassOf`, as discussed above, that would have a different implication. The restriction description under `rdfs:subClassOf` states a *necessary* condition while `owl:equivalentClass` goes a step further to create a *necessary and sufficient* condition. Thus, if the OWL statement in Figure 23 used `rdfs:subClassOf`, it would imply that individuals that has the `consistOf` property value of water may or may not be the same as `BodyOfWater` individual; however, the description using `owl:equivalentClass` declares that all things that consists of water must be a `BodyOfWater`. The definitions of `rdfs:subClassOf` and `owl:equivalentClass` are stated in Table 6.

Relationship	***Implication***
`rdfs:subClassOf`	BodyOfWater(a) implies consistsOf(a,b) & Water(b)
`owl:equivalentClass`	BodyOfWater(a) implies consistsOf(a,b) & Water(b) consistsOf(a,b) & Water(b)implies BodyOfWater(a)

Table 6. Construct for Necessary vs. Necessary & Sufficient Conditions

Similarly, properties may also use the construct `owl:equivalenProperty` to link properties together. That is, any two properties tied with this syntax have exactly the same value, or property extension.

e. Individual Equivalence Using owl:sameAs

Similar to the construct used for declaring that two classes are equivalent, the construct owl:sameAs is used to declare that two individuals are equivalent.. Figure 24 shows how the owl:sameAs construct is used to indicate that the individual `America` is identical to the individual `UnitedStatesOfAmerica`.

```
<Country rdf:ID="America">
  <owl:sameAs rdf:resource="#UnitedStatesOfAmerica">
</Country>
```

Figure 24. owl:sameAs Example

This syntax is also useful when linking multiple ontologies. `owl:sameAs` construct allows equating individuals from different OWL documents. The fact that individual equivalence or distinction is made explicitly implies that OWL does not assume uniqueness based on name. The above OWL statement asserts equivalence between two individuals. However, the same assertion may be inferred using a functional property. Given that the `hasCapitalCity` is a functional property, as defined above, the statement in Figure 25 states that the two individuals are equivalent.

```
<Country rdf:ID="UnitedStates">
  <hasCapitalCity rdf:resource="#DistrictOfColumbia"/>
  <hasCapitalCity rdf:resource="#WashingtonDC"/>
</Country>
```

Figure 25. Individual Equivalence Using Functional Property

Since `hasCapitalCity` is a functional property, it is simply inferred that `DistrictOfColumbia` is the same as `WashingtonDC`.

f. Individuals Differences Using owl:differentFrom & owl:AllDifferent

The inverse OWL construct of `owl:sameAs` is `owl:differentFrom`. This construct is used to make individuals explicitly distinct

from one another. This is important when using individuals as property values. Figure 26 illustrates a Geography example for the distinct individuals of `Climate`.

```
<Climate rdf:ID="Dry"/>

<Climate rdf:ID="TropicalHumid"/>
  <owl:differentFrom rdf:resource="#Dry"/>
</Climate>

<Climate rdf:ID="Highland"/>
  <owl:differentFrom rdf:resource="#Dry"/>
  <owl:differentFrom rdf:resource="#TropicalHumid"/>
</Climate>
```

Figure 26. owl:differentFrom Example

One way to assert distinction between individuals is to use the construct `owl:differentFrom`. By making these individuals explicitly distinct, it ensures that the properties do not assume equivalence for these individuals. That is, if a functional property tries to fill its value with two explicitly distinct individuals, an error would be raised. In the Geography ontology, the functional property, `hasClimate`, cannot have both `Dry` and `TropicalHumid` individuals as values of one instance. Since these two individuals are explicitly unique, by the `owl:differentFrom` construct, they cannot be made equivalent by a functional property.

Another method for declaring individual distinction is by using the `owl:AllDifferent` and `owl:distinctMembers` constructs[6].

```
<owl:AllDifferent>
  <owl:distinctMembers rdf:parsetype="Collection"/>
     <Climate rdf:about="#Dry"/>
     <Climate rdf:about="#TropicalHumid"/>
     <Climate rdf:about="#Highland"/>
  </owl:distinctMembers>
</owl:AllDifferent>
```

Figure 27. owl:AllDifferent & owl:distinctMembers Example

[6] The `owl:distinctMembers` can only be used in combination with `owl:AllDifferent`.

Figure 27 shows how to distinguish all the unique individuals in one OWL statement, rather than with each individual declaration. The statement in Figure 27 is semantically identical to the OWL statements in Figure 26.

8. Complex Classes

There are additional constructs used to create classes or *class expressions*. Class expressions are nested class descriptions, without the need for naming each "intermediate class" separately. Class expression allows for complex classes using set operations. This is done using anonymous classes or value restricted classes. Specifically, there are three types of set operations, namely union (`owl:unionOf`), intersection (`owl:intersectionOf`), and complement (`owl:complementOf`). These constructs can be thought of as the "and", "or", and "not" logical operators. Another method of creating complex classes is by *enumeration*, where a class is described by exhaustively listing the individuals using the `owl:oneOf` construct.

a. Set Operators (intersectionOf, unionOf, complentOf)

Using the set operations as a class description is closer to a "definition" than other class description discussed thus far. That is, the membership of class is wholly determined by the set operation specification. The set operator "intersection of", `owl:intersectionOf`, describes a class with individuals that belong to *all* specifications listed under the class description. One example from the Geography ontology using the `owl:intersectionOf` set operation is the `IslandCountry` class.

```
<owl:Class rdf:ID="IslandCountry">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="Country"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasLandType"/>
      <owl:someValuesFrom rdf:resource="#Island"/>
      <owl:allValuesFrom rdf:resource="#Island"/>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Figure 28. owl:intersectionOf Example

In Figure 28, IslandCountry is strictly the intersection of Country class and the set hasLandType property values of the Island individuals. Therefore, all individuals of IslandCountry class must belong as extensions of both of these specifications. Notice the use of the syntax rdf:parsetype=”Collection” within OWL’s intersection syntax. As discussed in the RDF section, it is used to exhaustively list membership of a class. This construct will be used with the other set operations as well.

The second set operation is the “union of”, with the `owl:unionOf` construct. Unlike `owl:intersectionOf`, the `owl:unionOf` operator describes a class with individuals that belong to *at least one* of the specifications listed under the class description.

```
<owl:Class rdf:ID="SaltyBodyOfWater">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Ocean"/>
    <owl:Class rdf:about="#Sea"/>
  </owl:unionOf>
</owl:Class>
```

Figure 29. owl:unionOf Example

In Figure 29, the rules on the union logic imply that the SaltyBodyOfWater class includes the individuals of both the Ocean and Sea classes. Since the union set operator is an “or” logic, the description above states that all the individuals of SaltyBodyOfWater are made up of individuals of Ocean or Sea.

The third operator is `owl:complementOf`. This construct is the logic “not,” where the class describes all the individuals that do not belong to the specified class extension.

```
<owl:Class rdf:ID="PhysicalGeography"/>

<owl:Class rdf:ID="PoliticalGeography">
  <owl:complementOf rdf:resource="#PhysicalGeography"/>
</owl:Class>
```

Figure 30. owl:complementOf Example

In the OWL statement in Figure 31, the members of the `PoliticalGeography` include all individuals that are not members of the `PhysicalGeography` class. Since this construct can include a very large set of members, it is often used in combination with other operators, as in Figure 31.

```
<owl:Class rdf:ID="NonSaltyBodyOfWater">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#BodyOfWater"/>
    <owl:Class>
       <owl:complementOf rdf:resource="#SaltyBodyOfWater/">
    </owl:Class>
  </owl:intersectionOf>
</owl:Class>
```

Figure 31. owl:complementOf & owl:intersectionOf Example

In Figure 31, the NonSaltyBodyOfWater class is an intersection of two classes, one named and one anonymous. The class description, using two set operators, states that individuals of this class must be of both BodyOfWater and NOT a member of the SaltyBodyOfWater class.

b. Enumerated Classes (`oneOf`)

Another method of defining a class is by direct enumeration of all of its members or individuals. Using the `owl:oneOf` construct, the class is described by exhaustively listing all the individuals that make up the class. No other individual, other than those included in the list, can be a member of the class.

```
<owl:Class rdf:ID="SaltContent">
  <rdfs:subClassOf rdf:resource="PhysicalDescriptor">
  <owl:oneOf rdf:parsetype="Collection"/>
     <SaltContent rdf:about="Salty"/>
     <SaltContent rdf:about="NotSalty"/>
  </owl:oneOf>
</owl:Class>
```

Figure 32. Enumeration Example

In Figure 32, the SaltContent class is defined by enumeration; listing all the members, Salty and NotSalty, of the class. In order for this definition to be valid, every individual must be declared correctly and they must all belong to a named class. It

is not required for the individuals to be declared as a member of the class being defined, although that is often the most logical.

D. CONCLUSION

This chapter covered all the RDF and OWL semantics necessary to develop a OWL-DL ontology. Understanding these constructs is crucial to building a semantically rich ontology for the Web or other Web-related applications. Although there are OWL generating ontology editors available, such as Protégé, without learning how and when the OWL semantics are used, it is not possible to build a valid ontology. This chapter should serve as the foundation to create a useful and meaningful ontology.

Having gained an understanding of all the basic components of the RDF and OWL, the next step is learning the process for developing an ontology. The next chapter describes a methodology of ontology development.

THIS PAGE INTENTIONALLY LEFT BLANK

III. ONTOLOGY DEVELOPMENT METHODOLOGY

A. INTRODUCTION

It is generally agreed upon that ontologies are the knowledge representation component of the Semantic Web. Although the realization of the Semantic Web is still a distant goal, there is a growing interest for ontologies to be incorporated into current technologies. Many disciplines are seeing the immense value of ontologies as a way to codify a common set of information or knowledge to be shared across multiple applications. It provides users with a consistent and agreed-upon knowledge base that both humans and machines can process. While no ontology can model all the nuances of any domain area, it is possible and valuable to abstract the major concepts and how they relate to one another. A valid knowledge representation system that is widely used and shared could save effort for those who lack access to subject matter experts (SMEs). Likewise, SMEs are motivated to provide users and applications with basic domain knowledge through the development of ontologies, thus providing users with consistent sets of information that they can maintain and manage.

Following the discussion of OWL-DL constructs presented in the previous chapter, this chapter examines a methodology for developing an OWL-DL ontology. This process involves modeling the real world concepts and their relationships into OWL classes, properties and instances. Although ontology development may be easily understood, ontologies should be developed by SMEs or domain experts, particularly in complex or technical domain areas, such as biomedicine or aircraft components. This chapter addresses an approach of transferring domain knowledge into a valid ontology. It is important for ontology developers to understand that building a meaningful ontology is a highly iterative process. The greater the complexity of the domain and its scope, the more iterations or cycles of development is required.

When an ontology is developed in OWL-DL, one can use a reasoning or an inference application to correctly classify the concepts of the domain based on their descriptions. As mentioned in Chapter 2, OWL-DL derives its semantics from *description language,* which is a description logic formalism for representing logical

meaning for reasoning computations. The capacity of inferencing is a highly valuable component, especially as the ontology grows in size and complexity. There are applications, such as CLASSIC and RacerPro, dedicated to processing the description logic languages. For ontology development in this chapter and the OAKDA application in the latter chapters, the RacerPro[7] will be the reasoning and inferencing engine of choice for OWL-DL ontologies.

Throughout this chapter, the authors use the Protégé ontology development environment to illustrate the process of building an ontology. Although there are various OWL-DL ontology development environments, Protégé provides a user-friendly plug-in for RacerPro and a graphic user interface (GUI) that hides the details of the OWL syntax from the developer. While it is crucial to understand the OWL-DL constructs in building a valid ontology, the purpose of this chapter is to understand the process and methodology rather than the syntax. Thus, the focus will shift away from the language constructs to the developmental steps and key concepts unique to developing an OWL-DL ontology.

The rest of this chapter is dedicated to understanding the purpose and method of developing an ontology. Section B will discuss why ontologies are important and useful as a knowledge representation system. Section C surveys previous work on methodologies for building ontologies. Section D details a proposed seven-step development methodology using Geography as the domain of interest. Section E discusses other relevant considerations, such as exporting existing ontologies.

B. THE PURPOSE OF BUILDING AN ONTOLOGY

The word ontology has its origin in the philosophy discipline. Ontology is defined in the Merriam-Webster dictionary as "(1) A branch of metaphysics concerned with the nature and relations of being; (2) A particular theory about the nature of being or the kinds of existents." [http://www.m-w.com/, January 2006] This definition pertains to a particular study of metaphysical philosophy concerning the nature of existence and its

[7] RacerPro is a DL inference system. RacerPro, which stands for Renamed ABox Concept Expression Reasoner, is a knowledge representation application using highly optimized tableau calculus for description logic expressions. It provides reasoning for multiple ABoxes and TBoxes.

experience. This concept is usually referred to as the “Big O” Ontology because it defines a named idea in philosophy. However, there is also what is known as the “little o” ontology, which began in the field of Knowledge Management and it is widely adopted in Information Technology and AI disciplines. Specifically, Thomas Gruber of the Knowledge Systems Laboratory at Stanford University first defined an ontology as “specification of conceptualization.” More explicitly, an ontology is “a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents. This definition is consistent with the usage of ontology as set-of-concept-definitions, but more general.” [Gruber, 1993, 199] Gruber’s definition is further expounded to state that the “little o” ontology has two parts, namely that (1) it describes and represents an area of knowledge, and (2) it defines “the common words and concepts of the description.” [Daconta et al., 2003, 186]

Using ontologies as a system of knowledge representation is an important contribution made by the field of Knowledge Management. Although there are other systems or framework for knowledge representation, ontologies provide a reusable, sharable and platform neutral knowledge construct. According to Gruber, many other knowledge systems are "isolated monoliths characterized by *high internal coupling* and a *lack of external coupling interfaces* that would enable the developer to reuse software tools and knowledge bases as modular components.” [Gruber, 1991, 1] Thus, he argues that the only possible method of sharing and reusing these knowledge bases is to import the knowledge representation system and its programming environment. However, using the software engineering approach of “decomposing” these indivisible systems, these knowledge bases should be broken into accumulable, sharable and reusable modular building blocks. Gruber states that three decomposition techniques are often used in AI for software development. These are using declarative knowledge representation, separate the knowledge from the program; identifying the classes and relationships inherent in the application-specific facts and reorganize the knowledge to allow inheritance from these constructs; and characterizing general problem solving tasks (i.e. classification) and inferencing classes (i.e. subsumption) and design corresponding algorithms and methods [Gruber, 1991, 2]. However, he further argues that these

strategies alone are insufficient to ensure sharability of the system. In addition to formalizing declarative knowledge, organizing class and relationship hierarchies, and characterizing tasks and inferences, Gruber introduces the need for *canonical form* of the declarative knowledge and *common ontologies*. First, "canonical form of the declarative knowledge" is a standard set of syntax and constructs or "semantics" that will be used as the knowledge representation language. Second, common ontologies are "vocabularies of representational *terms* – classes, relations, functions, object constraints – with agreed-upon *definitions*, in the form of human readable text and machine-enforceable, declarative constraints on their well-formed use." [Gruber, 1991, 2] The canonical form and common ontologies are two necessarily elements that allow the knowledge representation systems to be shared and reused across multiple platforms.

Based on Gruber's broad definition, an ontology can take on various forms. Types of an ontology may be as basic as a simple catalog, a finite list of terminology, and as semantically sophisticated as logical abstraction for disjointed and inverse relationships, shown in Figure 33

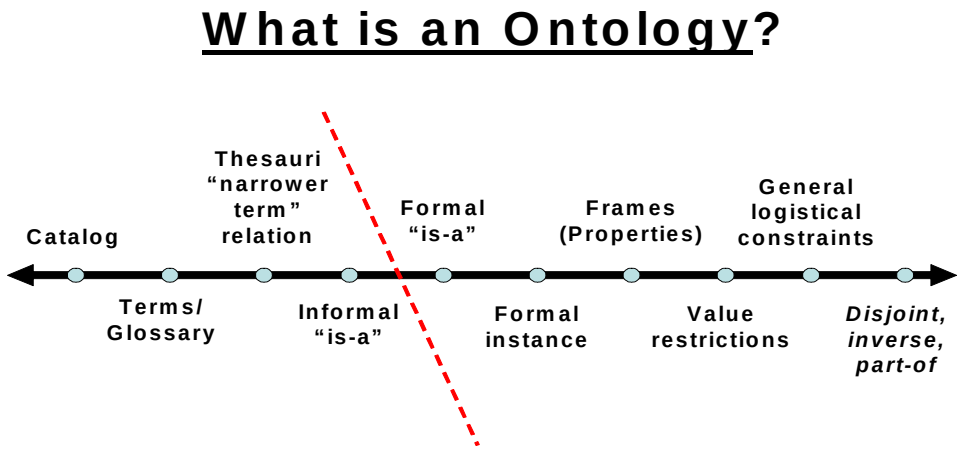

Figure 33. Ontology Spectrum

The Ontology Spectrum [McGuiness, 2001, 3] diagram shows the progression of concept organization. As ontologies move from simple taxonomies to a structured

knowledge base with properties and restrictions, their need for expressiveness grows. The right side of the dotted line in Figure 33 indicates where OWL-DL constructs become relevant. At the far right end of the spectrum, OWL-DL becomes imperative as an ontology language. Furthermore, an inference engine uses OWL's description logic capability to verify consistency and completion. It mathematically checks for consistency and makes inference where it deems the relationships to be incomplete. These are crucial elements of a meaningful ontology because applications and systems rely on valid knowledge representation. The ontologies of concern in this thesis fall in the farthest end of the Ontology Spectrum of Figure 33.

Given the wide range of ontologies, why are they important or even relevant as an information system? Natalya Noy and Deborah McGuinness argue five specific reasons for developing and using an ontology (Noy et. al., 2002, 3).

- *Share a common information structure* – By using an ontology that creates a common language amongst disparate systems, it becomes possible to share the same set of terms and concepts. This also allows agents to aggregate and extract information from other systems and use them appropriately to answer queries.
- *Reuse domain knowledge* – Reuse is an important benefit of ontologies because it allows separate components to join together seamlessly. By integrating existing ontologies, developers can rely on a trusted source of domain expert of a particular component. Rather than recreating something from nothing, existing ontologies can integrate into complex knowledge bases.
- *Make domain assumption explicit* – By incorporating domain assumptions into the ontology, rather than hard-coding it into a system, it makes it easier to manage and change them. It also helps "users" understand and learn the concepts and relationships within the domain.
- *Separate domain and operational knowledge* – The ontology, representing the domain knowledge, is disconnected from the application, which represents the operational knowledge. This kind of low coupling is valuable in managing change in complex systems.
- *Analyze domain knowledge* – The accessibility of ontologies makes it possible to analyze and validate domain knowledge. This is a key part of reusing and extending any ontology and is a valuable asset to developers and users alike.

In spite of all these reasons, representing real-world concepts and complex relationships in simplified two-dimensional ontologies is not only difficult, but experts

find it insufficient to model all the intricate relationships of a domain. Although OWL-DL has constructs richer in semantics than its predecessors such as RDF, it is still a modeling language for representing the real world concepts and relationships. Since semantic-richness must be juxtaposed with computability, there is a conflict between "transparency and predictability" and "rigor and completeness," when considering the design and development of an ontology [Rector, 2004, 5]. That is, according to Rector, the SME's preference for representing the domain in its practical way may not align with "logic and computational tractability." Hence, the developer must constantly balance the two perspectives. The tradeoff is between the domain expert's preference for rich representation of reality and the logics of calculation for the sake of inference and classification.

The goal of the ontology development is not only to represent the domain concepts and properties, but also to create a document that can be processed, including inferences, by machines. Therefore, rather than trying to model all the components of a particular domain, it is preferred to limit to scope to a particular area of interest or application. Before starting the development, the SME should first ask, how will the ontology be used? The answer to this question determines the scope of the ontology and the purpose of building it, which affect the overall design and structure of the ontology.

C. METHODOLOGIES FOR ONTOLOGY DEVELOPMENT

The growing number of ontologies developing in a variety of fields has lead to many proposed methodologies. All these different methods sprung from different domains and necessities, and they all bring important lessons for future developers. This section will review some of the widely known and used methodology for ontology developments.

1. Toronto Virtual Enterprise (TOVE)

The first known methodology derives from the experiences of Toronto Virtual Enterprise (TOVE) ontology development. The TOVE methodology steps are as follows:

1. *Motivating Scenarios* – These depict the set of problems facing an organization, which are described in scenario stories or examples.

2. *Informal Competency Questions* – Based on the scenarios from Step 1, these are informal questions that the ontology should be able to answer.
3. *Terminology Specification* – All the objects and their relationships are defined at this step in first order logic.
4. *Formal Competency Questions* – All the ontology terminologies are formalized.
5. *Axiom Specifications* – All the axioms that specify the object definitions and constraints are formally described. Theses axioms must satisfy and answer the competency questions stated in Step 4.
6. *Completeness Theorem* – This is the evaluation stage of the development, where the ontology is tested to meet all the required conditions.

It is argued that the most interesting aspect of the TOVE approach is its evaluation process using completeness theorem. The theorem are important for "assessing the extensibility of an ontology – any extension must be able to preserve the validity of the completeness theorems – or to provide a benchmark for ontologies" [Jones et al, 1998].

2. METHONTOLOGY

Similar to TOVE, METHONTOLOGY focuses on the assessment and maintenance of the ontology. The major difference between the two is that METHONTOLOGY focuses mainly on the maintenance within the life cycle whereas TOVE uses formal techniques for addressing limited areas of maintenance [Jones et al., 1998]. The seven steps of METHONTOLOGY are as follows:

1. *Specification* – This step states the purpose of the ontology as well as the users, application, scope, and the required level of formality. The output of this step is a "natural-language ontology specification document" [Gomez-Perez et al., 1995].
2. *Knowledge Acquisition* – In parallel with Step 1, the developer finds the source of the ontology domain knowledge in the form of interview with SMEs and analyses of literature.
3. *Conceptualization* – The terms of the domain are specified as concepts, instances, verb relations or properties.
4. *Integration* – For the sake of uniformity between different ontologies, specifications from other ontologies are consulted and incorporated.
5. *Implementation* – The ontology is developed into a formal language, such as OWL.
6. *Evaluation* – Significant attention is paid to this step of the methodology, using different techniques to determine the validity and verification of the ontology. A set of guidelines are used to search for incompleteness, inconsistencies, and redundancies.
7. *Documentation* – All the steps and the ontology life cycles are documented.

When the ontology is in the prototype phase, the emphasis is on the specification, conceptualization, formalization, integration and implementation steps of the lifecycle. However, after the ontology matures into the maintenance phase, the developer must shift the focus to knowledge acquisition, evaluation and documentation of the ontology [Jones et al, 1998].

3. KBSI IDEF5

The IDEF5 methodology proposes a set of "guidelines" rather than systematic rules for developing an ontology. It suggests that ontology engineering is an open-ended process that should be constantly refined and updated. The IDEF5 guidelines are as follows:

1. *Organizing and Scoping* – In the form of a purpose statement, this step establishes the objectives and context of the ontology that will be used as a "completion criteria."
2. *Data Collection* – The necessary data is collected using the knowledge acquisition techniques, such as expert interviews and protocol analysis.
3. *Data Analysis* – This step analyzes the data collected in Step 2. All the domain's objects of interests are identified and the boundaries of the ontology are defined.
4. *Initial Ontology Development* – A draft of the ontology is developed using "proto-concepts," which are preliminary specifications of objects, properties and relationships.
5. *Ontology Refinement and Validation* – The "proto-concepts" from Step 4 are tested and refined through multiple iterations using deductive validation methods.

Two representation languages assist in the continual refinement process of the IDEF methodology. The first is the schematic languages, which are graphical notations used mostly to facilitate the communication between the ontology developer and the domain expert. The second is the elaboration language, which is a more structured representation of the ontology objects and relationships [Jones et al., 1998].

The three ontology development methods presented above are equally valid and bring important lessons to ontology developers. However, this thesis proposes a new ontology development methodology that is unique to developing an OWL ontology. The goal is to incorporate the different methods proposed above as well as provide a useful step-by-step guidance using OWL as the knowledge representation language of the

ontology. Similar to three methodologies reviewed above, the following methodology is derived from the lessons learned by the authors while developing the Geography ontology.

D. THE STEPS TO DEVELOPING AN ONTOLOGY

Three basic approaches are used for developing ontologies: top-down, bottom-up, or a combination approach. Although no one method is best, the combination approach, that starts with identifying the most obvious concepts and then incorporating the less salient concepts later, is better aligned with the iterative process recommended in building an ontology, as well as other complicated systems or applications. The ultimate goal is to follow a process leading to a good design and proper structure of the ontology. Regardless of how well planned, ontology development should be a cyclical and iterative process. It is recommended that the developer consider the structure of the ontology in multiple iterations, similar to the life cycle approach of software development. An example of the life cycle approach for software development is the Boehm's spiral model, shown in Figure 34 (Boehm, 1998, 61). The model includes all the required phases of requirements, analysis, design, coding, testing and operations.

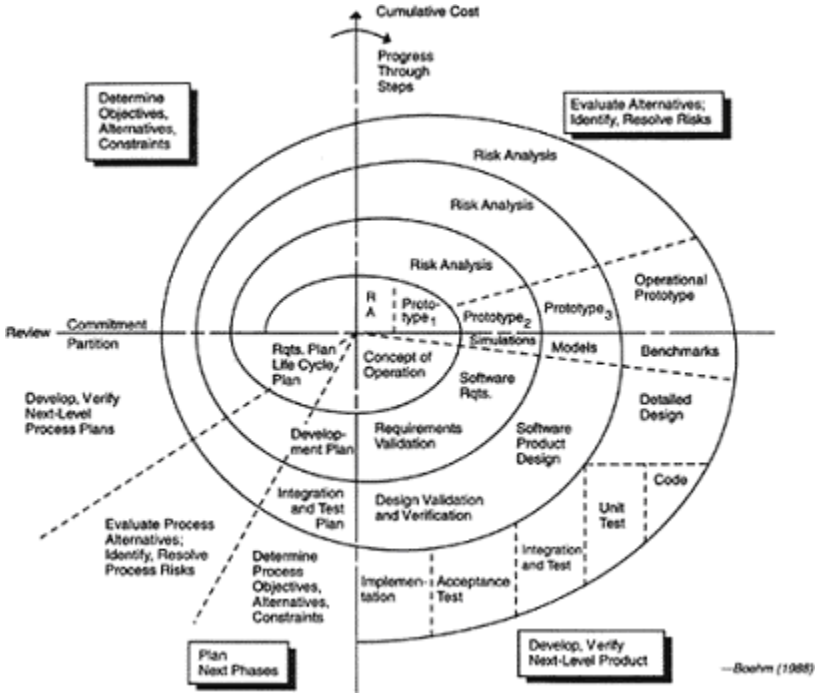

Figure 34. Boehm's Development Spiral

The steps outlined below fit into this lifecycle phases. While this chapter outlines the first iteration of the process, developers should expect to follow multiple repetitions of the methodology.

We propose an ontology development methodology that consists of seven steps. Similar to the spiral model, these steps are applied iteratively and the developers may find themselves going back to earlier steps to edit their initial work. The seven steps are as follows:

1. *Determine the scope and application of the ontology*
2. *List relevant concepts of the domain*
3. *Create the class hierarchy*
4. *Define properties*
5. *Describe classes using property restrictions and complex definitions*
6. *Classify ontology with a reasoning tool*
7. *Create individuals and fill property values*

Each of these steps will be discussed in detail in the sections that follow. These steps will be used to build the example Geography ontology presented in the previous chapter. The

choice of the Geography domain is based on the fact it is a commonly understood domain and thus will help the reader understand the process of building an ontology..

1. Determine the Scope and Application of the Ontology

This first step is the most important steps of the overall process and determines the final outcome of the ontology. The domain expert must understand well the purpose of the ontology. Often, the purpose of an ontology is two-folds. If an ontology is to represent the knowledge base of a particular domain or segment of a domain, it will potentially function to "answer" all general questions relating to that domain. A second reason for developing an ontology is their use as knowledge representations in specific applications. For a given ontology, the requirements for these two goals, to serve as a knowledge database for a specific application *and* as a generic knowledge representation model of a particular domain, may be conflicting. Therefore, the developer must compromise the demand for specificity and generality of scope in order to create a useful ontology. The developer should carefully manage the scope and depth to develop a realistic and coherent ontology that serves the purpose of the application.

To help developers determine the scope of a given ontology, a series of *competency questions* was developed by Gruninger et al. (Gruninger and Fox, 1995). These are questions that the ontology is expected to answer for the application on hand and, therefore, these questions can help determine the scope of the ontology. The list should include broad and specific questions, acting as the litmus tests to ascertain the necessary level of detail.

The scope and purpose of the Geography ontology is to define the basic physical and political geographies and represent the relationships between them for the purpose of using it with OAKDA application, which will mine it to provide meaningful context to tailor user web searches. It should represent the high-level understanding of geopolitics – the physical geographic characteristics existing within different types of political entities. We will use this example ontology in the sections that follow to demonstrate the development methodology of an OWL-DL ontology.

Example competency questions that are used to help determine the scope of the Geography ontology include the following:

- *Which non-democratic countries are landlocked?*
- *What bodies of water border all bi-coastal countries in the northern hemisphere?*
- *What is the most common climate of island countries?*
- *What river runs through the most countries?*
- *Which continent has the greatest number of coastal countries?*

These competency questions should be used to guide the development process and are applied repeatedly during the phases of the methodology as well as between iterations to ensure that the ontology fulfills its purpose.

2. List Relevant Concepts of the Domain

Once the scope is broadly defined, this step enumerates, in no particular order, the main concepts of the domain of interest. Although the final ontology may not necessarily include all the concepts defined during this phase, the developer should list as many relevant concepts as possible. At this point, one should not be concerned with overlapping concepts, the relationships between them, or their properties. The goal of this step is to create a comprehensive list of the concepts of the domain in preparation for the subsequent steps of development.

Although as indicated, the properties and relationships of concepts should not be considered in this step, bearing in mind the main properties of concepts and how one concept relates to another could generate useful ideas of other related concepts. Another useful technique is to group related concepts into relevant categories. However, these categories should not be too narrow, which introduces difficulties further along in the process. It is important to note that this is still an informal stage of the development, where the SME should be more concerned about generating ideas rather than hard-coding specific concepts into categories.

For the Geography example, the relevant concepts of the domain include the following:

`ocean`, `sea`, `lake`, `river`, `mountain`, `land`, `plains`, `valley`, `desert`, `tropics`, `climate`, `country`, `government`, `city`, `boundary`, `continent`, `language`, `ethnicity`, `latitude`, `longitude`, `archipelago`, `coastline`, `Mexico`, `South America`

There is no particular order to specifying the list and the concepts were generated from the competency questions. This step provides the input for the next step, which is building the class hierarchy. While not every concept from this stage becomes a class, having a large pool of concepts relevant to the domain makes the hierarchy development easier. As in the requirements analysis for software development, the time and thought invested into the first two steps of the methodology provide great benefits and rewards during the subsequent steps of the methodology.

3. Create the Class Hierarchy

This step creates a class hierarchy by relating classes through the subsumption construct "is-a" relationship. An "is-a" relationship indicates that a member of a subclass is also a member of the superclass as shown in Figure35.

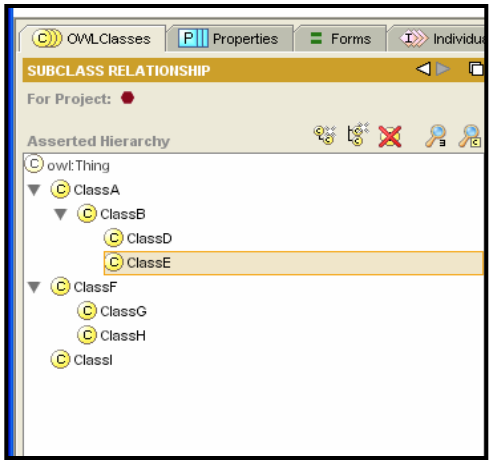

Figure 35. Simple Class Hierarchy

In the simple class hierarchy of Figure 35, classes `ClassA`, `ClassF` and `ClassI` are direct subclasses of `owl:Thing`[8], which is the highest OWL-defined class of the hierarchy. Classes within the same level of the hierarchy are considered *sibling*

8 As mentioned in the previous chapter, all classes in OWL-DL are subsumed under the parent class of owl:Thing. This implies that all classes are consider subclasses of owl:Thing. This is an important concept to remember as the ontology incorporates properties and domain and range restrictions.

classes. Classes subsumed under other classes are *subclasses*. All subclasses have a "is-a" relationship with their parent or superclass. Thus, members of `ClassB` are also members of `ClassA` according to the definition of the *is-a* relationship between `ClassA` and `ClassB`. Similarly, members of `ClassD` and `ClassE` are also members of `ClassB`, which in turn are members of `ClassA`. It is important to have a firm understanding of this parent-child class relationship to avoid problems with the later steps of the methodology.

Organizing the class hierarchy may be accomplished in several ways: top-down, bottom-up, or a combination approach [Noy and McGuiness, 2002, 6]. The top-down approach starts with the most general set of concepts and works down to the subsequent levels of specialization. For example, the `BodyOfLand` and `BodyOfWater` classes are identified as the highest level of the Geography ontology hierarchy, and subsequent subclasses are subsumed under these two classes. Thus, `Ocean`, `River`, and `Lake` are added as subclasses of `BodyOfWater`, and `Mountain`, `Desert`, and `Plains` as subclasses of `BodyOfLand`. The bottom-up approach starts with identifying the most specific classes, then grouping them under a parent class. In the Geography ontology example, the developer may start with `LandlockedCountry`, `IslandCountry`, and `BiCoastalCountry` classes, which are then grouped as subclasses under the parent class of `Country`. Similarly, lower level classes such as `DryClimate`, `PolarClimate`, and `TropicalHumidClimate` are subsumed under the `Climate` parent class.

When grouping low-level concepts, developers should carefully differentiate between classes and their instances, known in OWL as *individuals*. A careful examination of the list of concepts generated in step two of the methodology should help the developer differentiate classes from their instances. The distinction between a class and an individual is not always clear and often depends on the purpose of the ontology. This means that a concept that is a class in one ontology may be more appropriately represented as an individual in another. However, classes are generally "naturally occurring sets of things in a domain of discourse" and individuals correspond to real-

world entities grouped under these classes [Smith et al., 2004, 19]. Classes represent a group of similar entities while individuals are the actual occurrences of these entities that make up the group.

In the Geography ontology, `PacificOcean` is an instance of `Ocean` rather than its subclass since `PacificOcean` does not represent a group of entities but an actual entity itself. On the contrary, `IslandCountry` should be a subclass of `Country` rather than its instance since it represents a group of countries with the geographic landscape of an island, such as `Ireland` and `Cuba`. It is important to emphasize that the distinction between a class and an individual often depends on the purpose and scope of the ontology.

The most common approach to organizing the ontology class hierarchy is the combination approach. This approach develops the class hierarchy by defining the most salient terms of the ontology, adding successive classes at the different levels of the hierarchy as appropriate. The advantage of the combination approach is that it allows the developer to start anywhere along the hierarchy and move up and down the stratum to add new classes as necessary.

In the Geography ontology, two most salient classes are C`ountry` and `Ocean`. Using the combination approach, these two classes are defined at a top level of the hierarchy. Then, new concepts are added as parent classes or subclasses to these two initial set of classes. Furthermore, in line with the iterative development process, the hierarchy structure becomes refined as classes are moved from one position within the hierarchy to another. For example, as Figure 36 shows, in the first two iterations `BodyOfWater` and `BodyOfLand` classes occupied the top level of the hierarchy, as sibling to `PoliticalGeography` and `Climate`, however in the third iteration of the class hierarchy, the two classes fall under the parent class of `PhysicalGeography`.

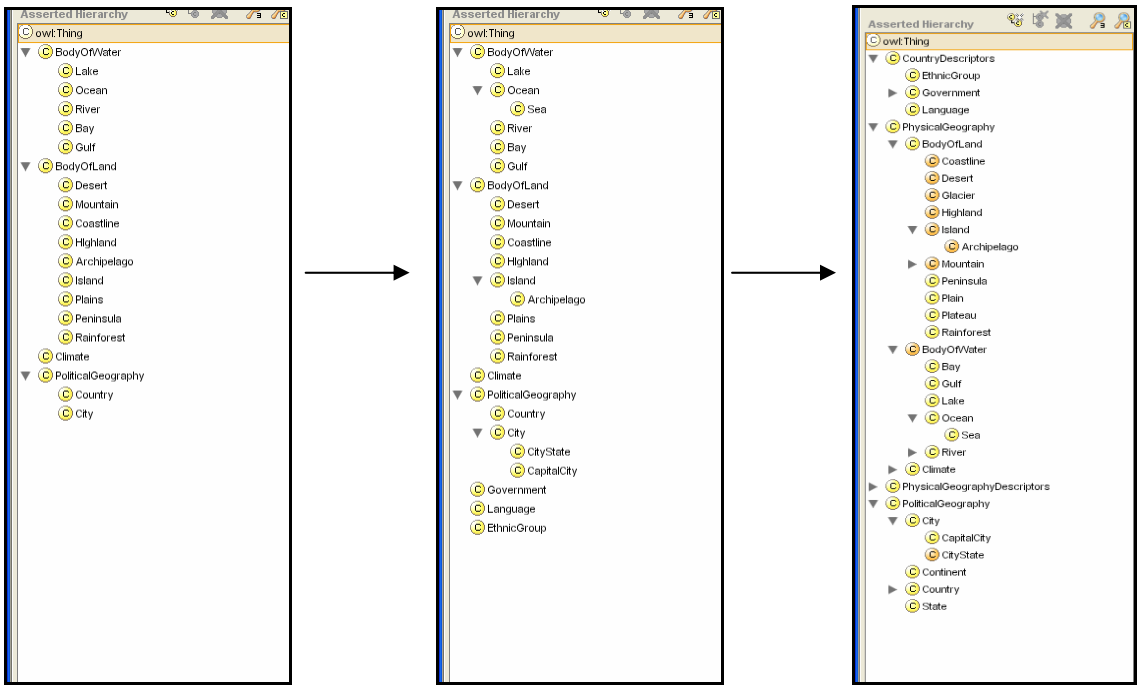

Figure 36. Progression of the Class Hierarchy

The development of the class hierarchy as described in this step falls under the "design" phase of the spiral development cycle. As the ontology evolves, the developer will revisit this step and modify the hierarchy as necessary. Additional requirements and knowledge acquired in the process refines the class taxonomy. In order to manage the constantly evolving ontology, detailed documentation and versioning is recommended.

a. Disjointed Classes

It is common for developers to make false assumptions about the relationship between OWL-DL classes. Specifically, developers presume that classes that do not share a superclass-subclass, or *is-a*, relationship are automatically disjoint. In OWL-DL, all classes are considered overlapping unless such separation or disjointness is made explicit. Specifying disjointness between classes requires an explicit specification using the OWL syntax `owl:disjointWith`. Only by defining a class as disjoint with others, the developer can assume class mutual exclusivity.

In the Geography ontology, disjointness between classes `Ocean`, `Mountain`, and `Country` is not assumed. In other words, according to the current specification, `Ocean` and `Mountain` classes can share the same individuals. In order to make classes disjoint from one another, disjointness must be specified explicitly using the `owl:disjointWith` statement.

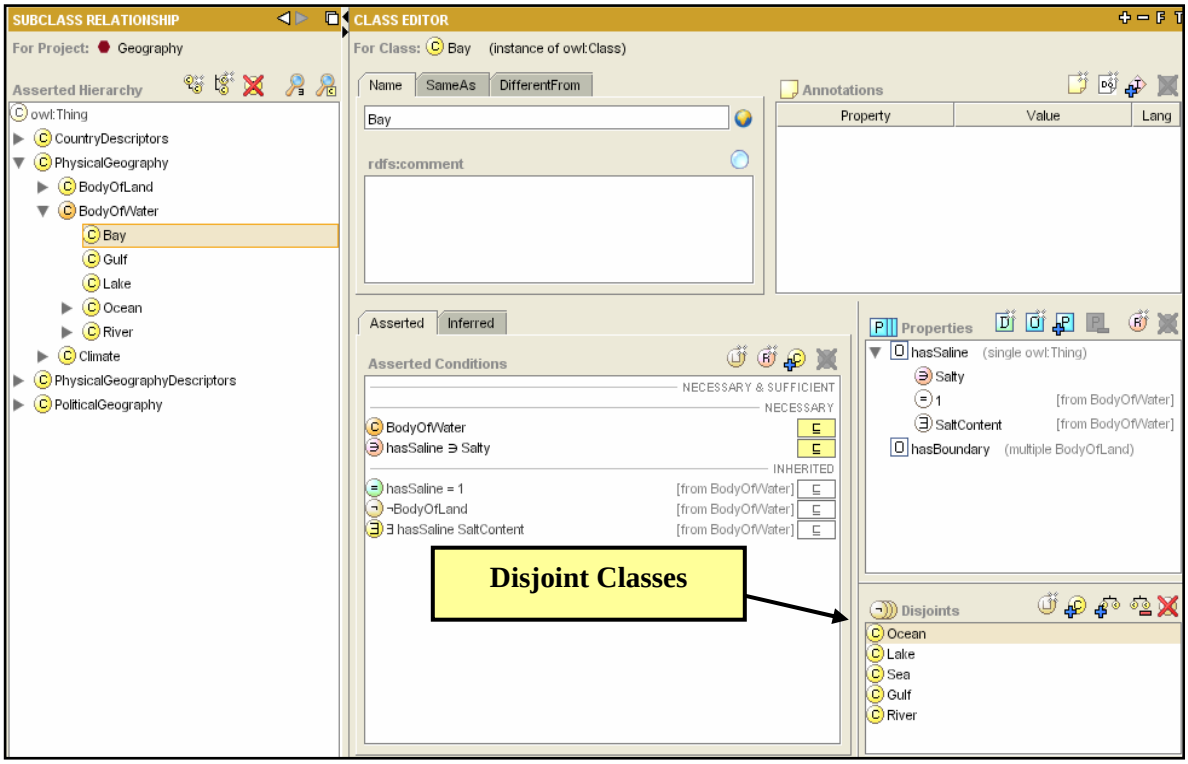

Figure 37. Disjointed Classes

Figure 37 indicates in Protégé that the selected `Bay` class is disjointed from all of its sibling classes, as listed in the bottom right corner box. Without this explicit restriction, OWL does not exclude an individual from belonging to more than one class. Also, the disjoint restriction applies to all the subclasses under the specified class. By making `BodyOfWater` disjoint with `BodyOfLand`, all the subclasses of `BodyOfWater` are disjoint from all the subclasses of `BodyOfLand`.

4. Define the Properties

After defining the class hierarchy, the next step is to specify the class properties. Classes, without any properties or restrictions, have no useful meaning other than how

they relate to each other in the taxonomy. A property, which is defined at the class level, represents the relationship between two individuals, or between an individual and a literal string value. As discussed in the previous chapter, depending on whether it is an object property or datatype property, the range value of the property is either an OWL individual object or a literal string. Properties describe the relationships or links between real-world objects or values. Properties are the verb that link the subject and object in the RDF triples as explained in Chapter Two. Although there are properties that are unique to a given class, it is more common and recommended that properties be defined generically and be applied to classes as appropriate. That is, a property often applies to more than one class. However, property restrictions can be used to limit the applicability, using the domain and range specification. This will be discussed in more detail in Section 5.C that defines the use of domain and ranges.

Although properties may be used generically throughout an ontology, the developer should start defining them based on the characteristics of the classes. One way of thinking about these characteristics is the verb-object that applies to the class. For instance, for the class `Parent`, the most obvious property is "has child." Likewise, the most salient property of the `Child` class is "has Parent." Similar to the technique used to define classes, developers should start with the most obvious characteristics of a class and iteratively add, change, and refine these characteristics.

In the Geography example, the characteristics of the `Country` class include "has border", "has population", "has capital", "has language", "has climate", "has river", "has lake", "has mountain", " has government", "has ethnic group", and others. As the list shows, most these characteristics relate to other classe instances within the ontology. The `Country` class has a `hasCapital` property to denote the relationship it has with a capital city. Although the verbs can be arbitrary and are often specified at the discretion of the developer, it is advisable to use the most straightforward and direct description of the relationship. For the class characteristics that relate it to a data type, the property depicts the class's relationship to a data string value. The property `hasPopulation` describes the link between the individuals of class `Country` and their population value. In this case, population is a numeric value that represents the number of people in a

particular country. Figure 38 shows the graphical representation of the difference between object oroperty and datatype property.

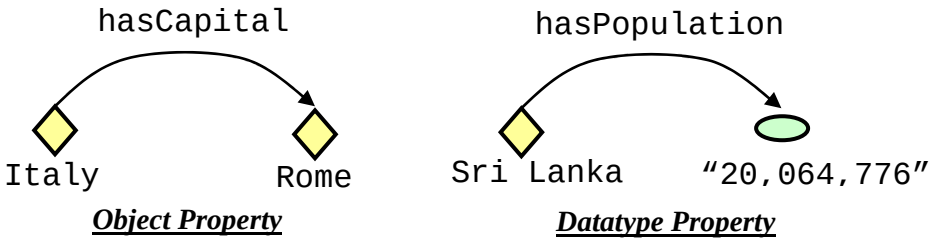

Figure 38. Two Types of Properties

The object property `hasCapital` denotes the relationship between the individuals of `Country` and the individuals of `CapitalCity`, which are both OWL objects. The datatype property `hasPopulation`, however, links the individuals of `Country` to a unique data string value, which in this case is "`20,064,776`."

A partial list of properties for the Geography ontology is shown in Figure 39.

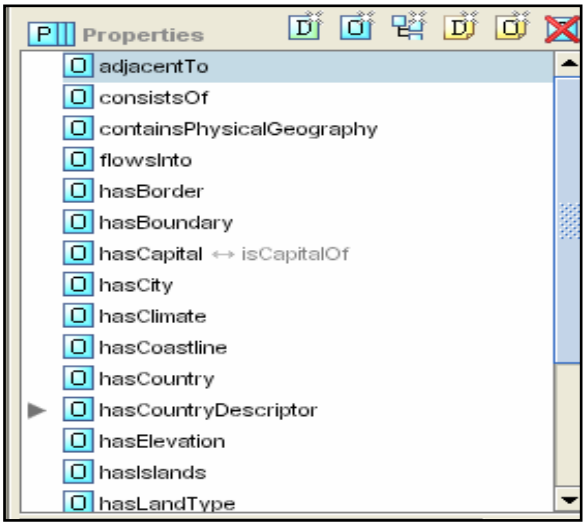

Figure 39. Geography Properties

OWL-DL ontologies allow the specification of different types of object properties. They include inverse, transitive, symmetric, functional and inverse functional properties. Each of these properties consists of its unique OWL-DL constructs, which is

discussed in detail in the sections below. It is important for the developer to correctly identify the type of property and specify it in the ontology.

a. Inverse Properties

Properties having an opposite relationship to one another are known as inverse properties. An inverse property is denoted using the OWL syntax, `owl:inverseOf`, a subproperty of `owl:ObjectProperty`, to indicate a diametric relationship to the specified inverse property. Generally, a property denotes a one-direction relationship from subject to object, such as `IndividualA` "`isParentOf`" `IndividualB`. Logically, `isParentOf` property, by itself, reveals no information about whether there is a corresponding relationship in the other direction. Developer can create another property, called `isChildOf`, to assert an opposite relationship from `IndividualB` to `IndividualA`, by designating the property as the inverse property of `isParentOf`.

Consider the Geography example in Figure 40.

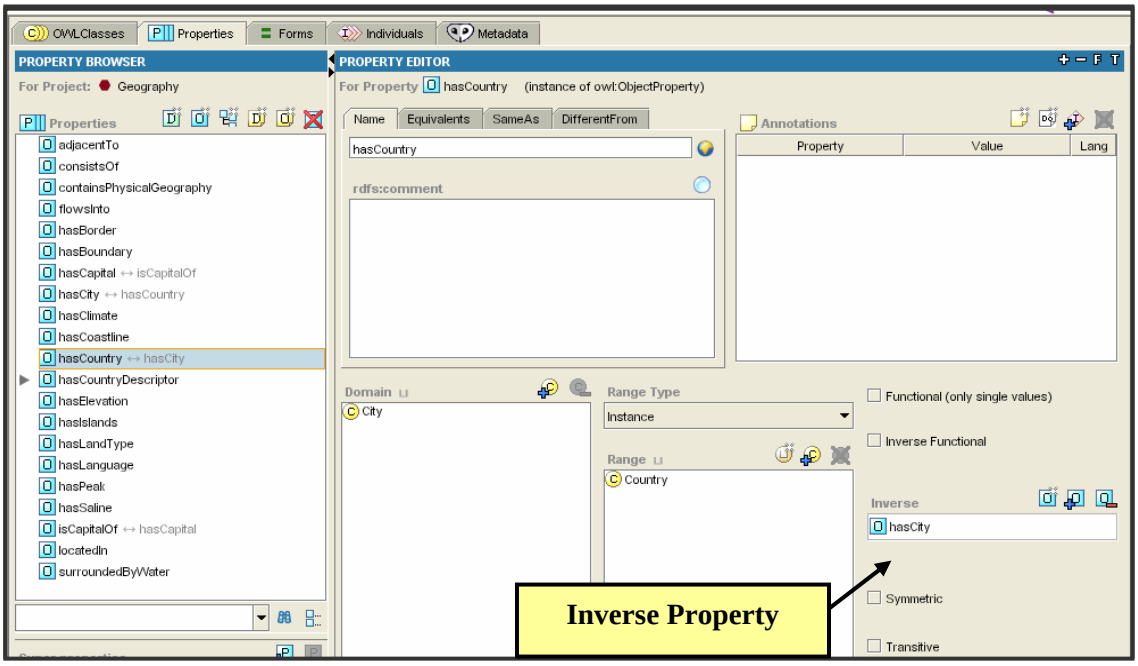

Figure 40. Inverse Property

The selected `hasCountry` property is an inverse property of `hasCity`, which is shown under the "Inverse" specification slot. If an individual has a

hasCountry property value filled by another individual, then by the rule of inverse property those two individuals also have an opposite relationship via hasCity property. That is, if the individual Venice, an instance of the City class, has the hasCountry property value of Italy, an instance of Country, Italy will automatically have the hasCity property value of Venice as shown in Figure 41.

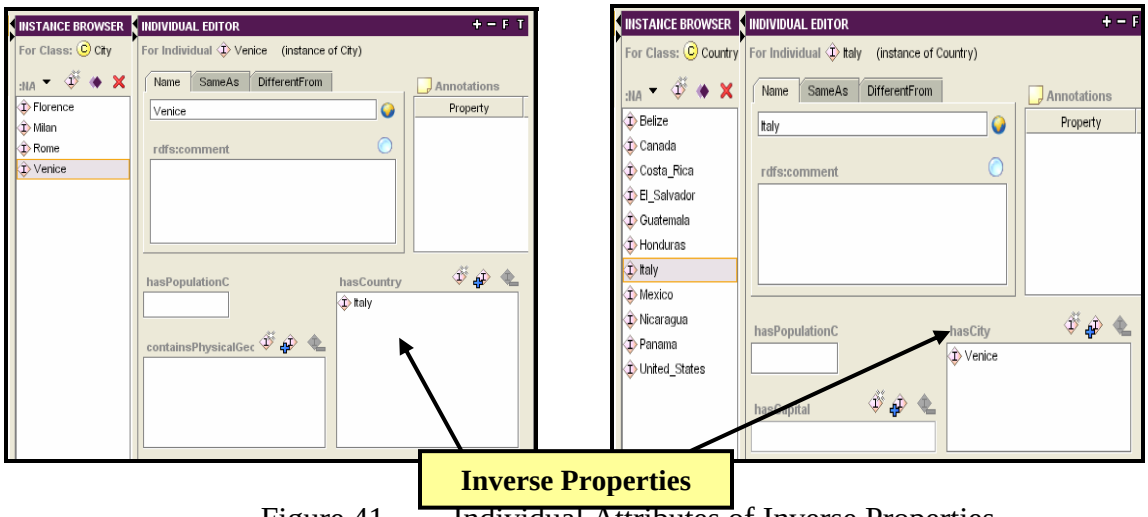

Figure 41. Individual Attributes of Inverse Properties

When using the inverse property, the domain and range axioms should be carefully considered. Although the inverse property example shown in Figure 8 has the domain and range defined, it is equally valid to leave them undeclared, which defaults to the highest class owl:Thing. In fact, if domain and ranges specification are not compatible between inverse properties, it may cause an error in the ontology and lead to unintended consequences.

b. Transitive & Symmetric Properties

Similar to the inverse property, transitive and symmetric properties are subclasses of owl:ObjectProperty and they assert information about the relationship of the individuals related by these properties. A transitive property is commonly used to represent "part-whole" relationships. That is, if transitive property P_T links individuals X and Y as well as individuals Y and Z, then it is inferred, by the rules of

transitivity, that P_T relates X to Z. Figure 42 shows how Protégé defines transitive and symmetric properties.

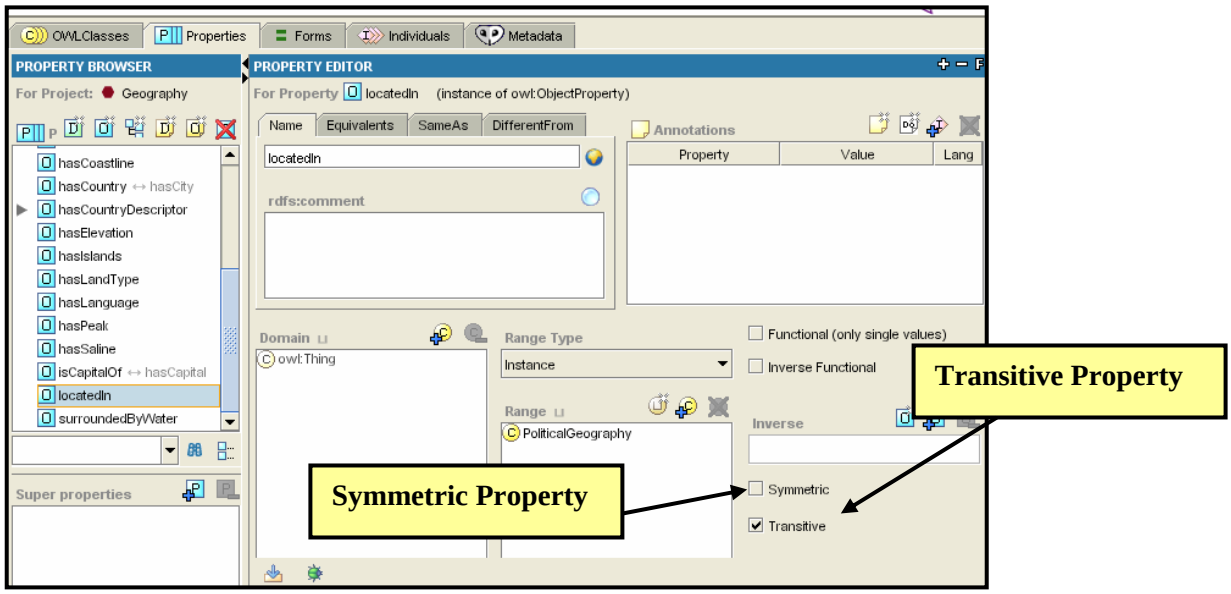

Figure 42. Transitive & Symmetric Properties

In the Geography ontology, the `locatedIn` is a transitive property and it is applied the individuals `VaticanCity`, `Rome`, and `Italy`. That is, if `VaticanCity` is "`locatedIn`" `Rome` and `Rome` is "`locatedIn Italy`", then by the rule of transitivity, `VaticanCity` is "`locatedIn`" `Italy`. While this implication is not explicitly stated in OWL or visible in Protégé, the inferred relationship is made transparent when the ontology is used to make reasoning decisions. Inference engines, such as RacerPro, read the OWL syntax and make the implied link as defined by the transitive property.

A symmetric property, on the other hand, allows the individuals to have a reciprocal or a bi-directional relationship. Unlike the inverse properties, a symmetric property is one relationship that is applied in both directions as shown in Figure 43.

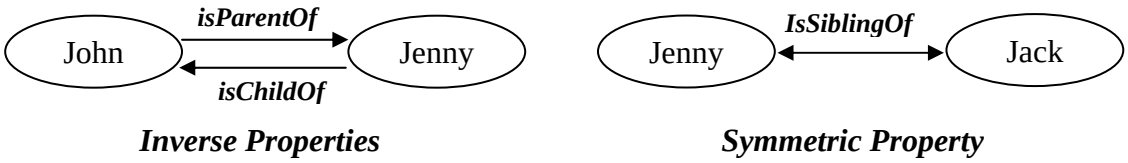

Figure 43. Difference between Inverse and Symmetric Properties

Figure 43 shows the difference between inverse and symmetric properties. The `isParentOf` property and `isChildOf` properties denote opposite relationships, making them inverse properties. However, the symmetric property `isSiblingOf` allows the relationship to be bi-directional, allowing the subject to be the object and vise versa.

In the Geography ontology, an example of symmetric properties is `adjacentTo`. When individual `A` is `adjacentTo` individual `B`, then by rule of symmetry, individual `B` is `adjacentTo` individual `A`. Specifically, if individual `Mexico` is `adjacentTo` `UnitedStates`, then it is inferred that `UnitedStates` is `adjacentTo` `Mexico`.

c. Functional & Inverse Functional Properties

A functional property indicates that, for a given individual, there can be at most one property value associated with that individual. For a functional property P_F, individual `X` is associated with at most one property value of individual `Y`. However, if P_F links `X` with another value, say individual `Z`, then by the rule of functional property, individual `Y` must equal individual `Z`. In other words, they are the same object or value with two separate instantiations.

Consider the example of Figure 44 from the Geography ontology. In this example, the property `hasCapital` is a functional property.

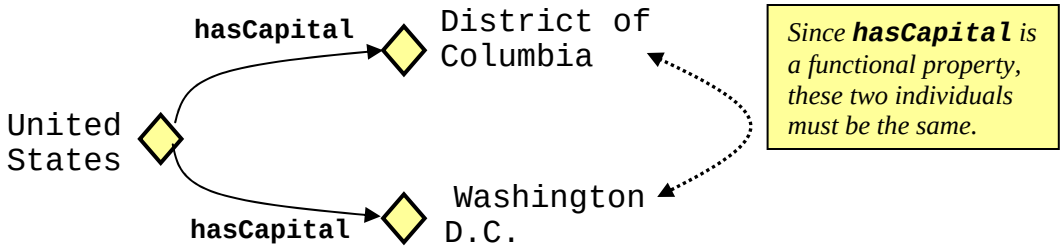

Figure 44. Attributes of a Functional Property

The individual `UnitedStates` is associated with two different `hasCapital` values, namely `DistrictOfColumbia` and `WashingtonDC`. However, since functional property must only have one value associated with a given individual, it must be inferred that these are equal objects.

Similar to inverse property, inverse functional property denotes an opposite relationship to its "inverse-of" property, which is a functional property. Since functional property is restricted to one property value, the same is applied to the inverse functional property. For an inverse functional property P_{IF}, if individuals X relates to individual Z and individual Y also relates to Z, then it is assumed that individual X equals to individual Y. An example of the inverse functional property from the Geography ontology is `belongsToCountry` property as shown in Figure 45.

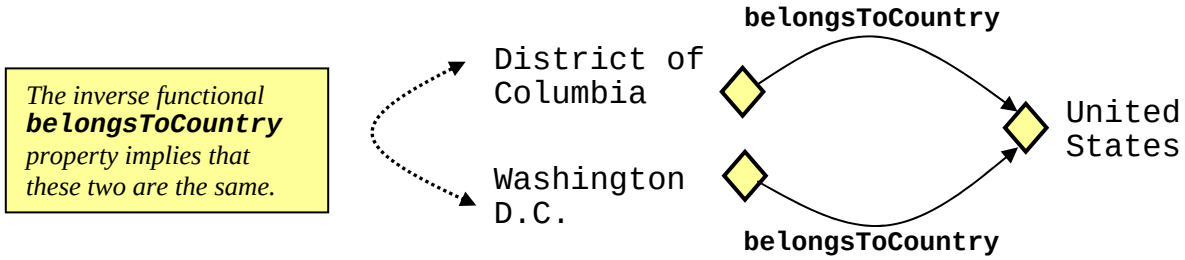

Figure 45. Attributes of an Inverse Functional Property

Figure 45 shows the values `DistrictOfColumbia` and `WashingtonDC` are both associated to `UnitedStates` by `belongsToCountry` property. By the rules of inverse functionality, it is inferred that these are equal and they are two instantiation of the same value.

Figure 46 shows how the functional and inverse functional properties are designated in Protégé.

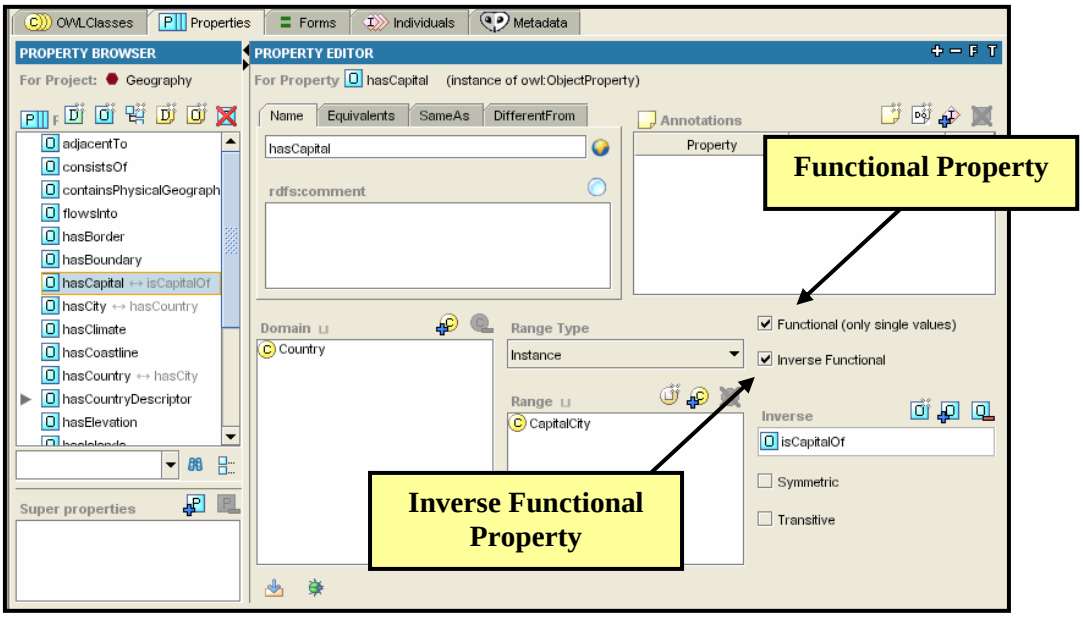

Figure 46. Functional and Inverse Functional Properties

Figure 46 shows that the definition of `hasCapital`, which is a functional property, specifies that it has an inverse functional property relationship to `isCapitalOf`. Once two properties are linked by the inverse-of construct, the specification only needs to be stated once; in this case the `isCapitalOf` property definition will automatically show that it has an inverse functional relationship with `hasCapital`.

5. Describe Classes Using Property Restrictions and Complex Definitions

Once properties are defined, they are used to restrict and describe classes. In order to associate a property with a class definition, it must be used as part of the class restriction. There are three types of class descriptions in OWL-DL, namely enumeration, property restriction, and complex class definition. First, enumeration describes a class by exhaustively listing all of its members or instances in its definition using the OWL construct `owl:oneOf`. No other members, other than those listed under the definition can belong to the class. Second, there are two types of property restrictions, *quantifier*

(or value) and *cardinality*. The quantifier restrictions constrain the range value of the property when applied to the class definition. The cardinality restrictions constrain the number of property values the class instance is allowed. And third, complex class descriptions are defined using logical operators of intersection (AND), union (OR) and complement (NOT) of classes. They represent advanced class logic of OWL-DL. Along with describing the restrictions in detail, there are two OWL-DL concepts, the difference between universal and existential restrictions and understanding "open world" vs. "closed world" assumption, that frame the types of restrictions used for class descriptions. These will be discussed in detail below.

a. Universal and Existential Restrictions

One of the most common errors when using property restrictions to describe classes is the differentiating between universal and existential restrictions. Without understanding the meaning and implications of these two restrictions, it is likely that many developers will use the wrong restriction. To constrain the range value of a property, an existential restriction (`someValuesFrom`) should be used rather than a universal restriction (`allValuesFrom`). The existential restriction, denoted with the symbol "∃", states that the individuals of the class being defined must have *at lease one* property relationship with the specified range of individuals. In other words, if a property restriction for `ClassX` is "$\exists$ $Property_E$ $ClassY$", then every individual of `ClassX` have *at least one* $\texttt{Property}_E$ relationship with an individual of `ClassY`. By this definition, however, it is possible to for individuals of `ClassX` to have $\texttt{Property}_E$ relationship with individuals of other classes as long as it satisfies the "at least one" requirement. It does not restrict the individuals to have $\texttt{Property}_E$ relationship with only the individuals of `ClassY`. On the other hand, universal restriction, denoted with the symbol "∀", states that individuals of the class being defined must have *all* of their property relationships with the specified range of individuals. For `ClassX` with a property restriction of "$\forall$ $Property_U$ $ClassY$", if individuals of `ClassX` have any $\texttt{Property}_U$ relationship, it must be with individuals of `ClassY`. However, it is possible for individuals of `ClassX` to not have any $\texttt{Property}_U$ values. Unlike the

existential restriction, universal restriction does not require the individuals to have any property relationship with a defined set of objects.

In description logic, existential restrictions are used to limit the property range, requiring every individual of that class to have at least one property value from the specified range. In defining the `Country` class, the property `containsFeatures` uses the existential restriction `someValuesFrom` as shown in Figure 47[9].

OWL:
Class (`Country`) Subclass of `PoliticalGeography`
Restriction (`containsFeatures someValuesFrom BodyOfLand`)

Translation:
`Country` class contains, *amongst other things*, some form of `BodyOfLand`

Figure 47. Existential Restriction in OWL

The existential restriction, "∃ `containsFeature BodyOfLand`", requires that at lease one of the `Country` individual have a `containsFeature` property value from the individuals of `BodyOfLand`. As long as that requirement is satisfied, individuals of the `Country` class may have `containsFeature` property value from individuals from other classes, as shown in Figure 48.

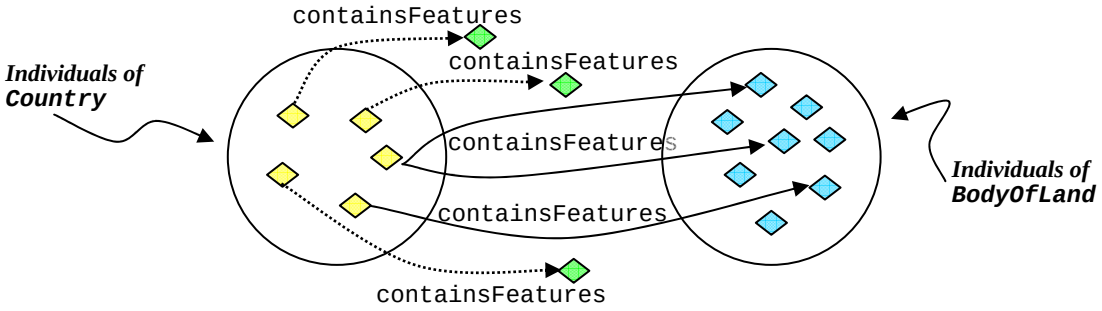

Figure 48. Existential Restriction Example

If the `Country` class was defined by a universal restriction, the `allValuesFrom` semantic is used to constrain the class description (Figure 49).

[9] The "translation" is the English paraphrasing of the OWL-DL semantics stated in the examples.

OWL:
Class (`Country`) Subclass of `PoliticalGeography`
Restriction (`containsFeatures allValuesFrom BodyOfLand`)

Translation:
`Country` class contains, *amongst other things*, some form of `BodyOfLand`

Figure 49. Universal Restriction in OWL

The universal restriction, "∀ `containsFeature BodyOfLand`", requires that if an individual of `Country` has a `containsFeature` property value, it must be an individual of `BodyOfLand`. However, this restriction does not require all of `Country` individuals to have a `containsFeature` property value. Unlike the existential restriction, individuals may not be associated with any `containsFeature` relationships. This is shown in figure 50.

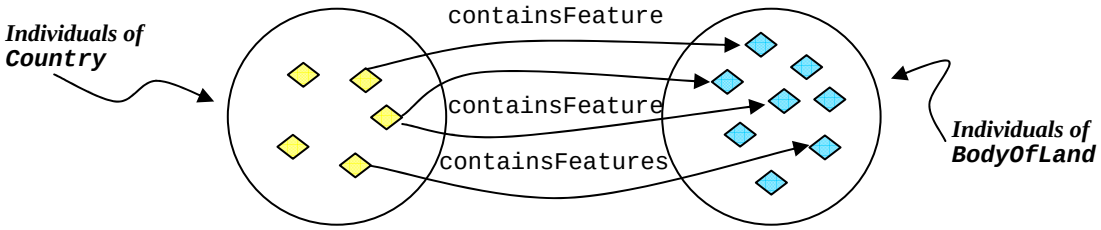

Figure 50. Universal Restriction Example

Developers should be clear about the appropriate type of property restrictions that should be applied to the class definitions. If the wrong restriction is applied to the class, there will be unforeseen consequences when the ontology is inferenced and affect the overall validity of the ontology.

b. Open World vs. Closed World

Most ontology developers, unfamiliar with open world reasoning of OWL, fail to make negation explicit. Databases, logic programming and frame languages are “closed world reasoning" systems which assume that when something is not found, it is false. However, description logic based languages, such as OWL-DL, associate negation

with “unsatisfiability.” That is, falsification can only be proven if contradicting information is made explicit.

Consider the `IslandCountry` and `LandlockedCountry` class definitions in Figure 51.

OWL:
Class (`IslandCountry`) Subclass of `Country`
Restriction (`hasLandType someValuesFrom Island`)
Restriction (`hasBorder someValuesFrom Ocean`)

Translation:
`IslandCountry` is any country that has, *amongst other things*, some land type of `Island` and some border of `Ocean`.

OWL:
Class (`LandlockedCountry`) Subclass of `Country`
Restriction (`hasBorder someValuesFrom Country`)
complementOf (restriction (`hasBorder someValueFrom Ocean`))

Translation:
`LandlockedCountry` is any country that has, *amongst other things*, some border of `Country` and does not have some border of `Ocean`.

Figure 51. Definitions of IslandCountry and LandlockedCountry

Intuitively, developers will define the classes `IslandCountry` and `LandlockedCountry` as stated in Figure 51. When a developer uses the existential restriction statement `someValuesFrom`, to restrict the property value, it leaves the definition open to other assumptions, as expressed by the translation “*amongst other things*." That is, these definitions are open to interpretation that the individuals can have property values other than what was specified with the `someValuesFrom` restrictions. Although these definitions are not technically invalid and do not cause problems by themselves, complications occur when you introduce other classes, such as `ArchipelagoCountry`, to the ontology. Consider the definitions in Figure 52.

OWL:
Class (`Archipelago`) Subclass of `Island`
Restriction (`cardinality >2 Island`)

Translation:
`Archipelago` is any island that consists of, *amongst other things*, at least two islands.

OWL:
Class (`ArchipelagoCountry`) Subclass of `Country`
Restriction (`hasLandType someValuesFrom Archipelago`)

Translation:
`ArchipelagoCountry` is any country that has, *amongst other things*, some land type of `Archipelago`.

Figure 52. Definitions of Archipelago and ArchipelagoCountry

Based on the specifications of `Archipelago` and `ArchipelagoCountry`, it may seem appropriate that `ArchipelagoCountry` class be subsumed under `IslandCountry` class, since `Archipelago` is a subclass of `Island`. However, given the open definition of `IslandCountry`, as shown in Figure 19, `ArchipelagoCountry` will not be inferred or classified as an `IslandCountry`. The current definition of `ArchipelagoCountry`, described by the existential property restriction of `someValueFrom`, does not preclude the class from having a land type of something other than an `Archipelago`. Thus, `ArchipelagoCountry` can take on any form of land type and be classified as a `LandlockedCountry` as likely as any other type of country in the ontology.

Based on the definition shown in Figure 20, open world reasoning makes no assumptions about the land type of the `ArchipelagoCountry` class simply based on the fact that other land type information was absent from the definition. In order for this class to be classified as a subclass of `IslandCountry`, as is the intention of the developers, it must explicitly exclude of all other land types in the class definition. This is accomplished by including a further restriction known as a *closure axiom*. According to Horridge et al., "[a] close axiom on a property consists of a universal restriction that

acts along the property to say it can only be filled by the specified fillers [Horridge et. al., 2004, 70]. The restricton has a filled that is a union of the fillers that occur in the existential restriction for the property." That is, closure axiom adds the universal restriction `allValuesFrom` to the existing existential restriction to exclude other possible assumptions in the class definition. Hence, it closes the class definitions to other interpretations, as shown in Figure 53.

OWL:
Class (`ArchipelagoCountry`) Subclass of `Country`
Restriction (`hasLandType someValuesFrom Archipelago`)
Restriction (`hasLandType allValuesFrom Archipelago`)

Translation:
`ArchipelagoCountry` is any country that has, *amongst other things*, some land type of `Archipelago` and ***only*** land type of `Archipelago`.

Figure 53. New Definition of `ArchipelagoCountry`

By applying the closure axiom, the definition of `ArchipelagoCountry` is no longer open or ambiguous. As written under the translation, the `allValuesFrom` specification adds the restriction “only” to the definition, disallowing the `hasLandType` property from including any individuals other than those belonging to the `Archipelago` class. The closure axiom should be applied to all classes where such quantifier property restrictions apply; otherwise, inferencing tools cannot properly classify the classes. Hence, classes `IslandCountry` and `LandlockedCountry` should also include closure axioms as shown in Figure 54.

OWL:
Class (`IslandCountry`) Subclass of `Country`
Restriction (`hasLandType someValuesFrom Island`)
Restriction (`hasLandType allValuesFrom Island`)
Restriction (`hasBorder someValuesFrom Ocean`)
Restriction (`hasBorder allValuesFrom Ocean`)

Translation:
`IslandCountry` is any country that has, *amongst other things*, some land type of `Island` and some border of `Ocean` and only land type of `Island` and only border of `Ocean`.

OWL:
Class (`LandlockedCountry`) Subclass of `Country`
Restriction (`hasBorder someValuesFrom Country`)
Restriction (`hasBorder allValuesFrom Country`)
complementOf (restriction (`hasBorder someValueFrom Ocean`))
complementOf (restriction (`hasBorder allValuesFrom Ocean`))

Translation:
`LandlockedCountry` is any country that has, *amongst other things*, some border of `Country` and does not have some border of `Ocean` and only has border of `Country` and never has border of `Ocean`.

Figure 54. New Definitions of IslandCountry and LandlockedCountry

c. Domain and Range

Each property has an associated domain and range as part of the property definition. When the property is initially created, its domain and range defaults to OWL's highest level class, the `owl:Thing` class. However, the developer may change the domain and range to other values. As mentioned in the previous chapter, the domain of a property associates the property with the class(es) it modifies and asserts that the subjects of such property statements must belong to the instance of the class. The range is the specified set of values, either class(es) or data string, that the property is allowed to take as its value. For an object property, the property links the individuals of the domain class to the individuals of the range class. Unlike the quantifier restrictions, domain and range are global axioms that are applied wherever the property is used, rather than only at the class description level.

If the Geography ontology's `hasCapital` has a domain of `Country` and range of `CapitalCity`, then this property is intended to connect the individuals of

`Country` to the individuals of `CapitalCity` whenever the `hasCapital` property is used as a restriction.

When specifying an object property, it is important to define the range type of "instance," rather than "class," which is the other possible option in OWL (Figure 55). Developers commonly mistake the range values of a property to be the class objects themselves, rather than the individual(s) that belong to that class. By choosing "class" as the range type, OWL will treat the class as an individual creating a type of "meta-statement" allowed only in the OWL-Full sublanguage.

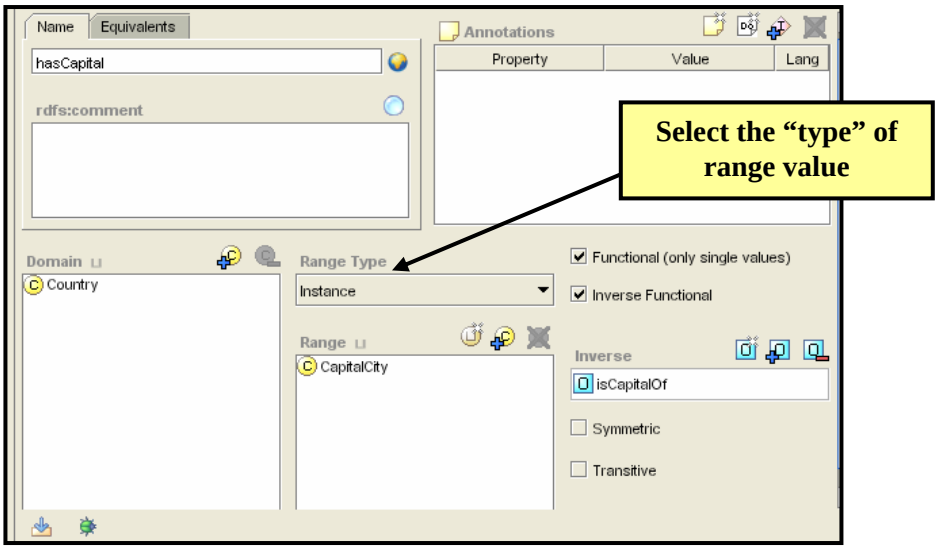

Figure 55. Selecting the Property Range Type

Range of an object property may consist of individuals from more than one class. If propertyP has a range of individuals from ClassA and ClassB, then the possible values of propertyP include all the individuals, or the union, of Class A and Class B. In the Geography ontology, the property `hasJurisdiction` is a property that links the individuals of the `Government` class to the individuals of both `City` and `Country` classes, via the property. When multiple classes are designated as the range, the OWL represents the range values as the union of classes, in this case the individuals that are either individuals of `City` or individuals of `Country`.

In OWL, domain and range constraints are axioms[10] used for reasoning, rather than binding restrictions of the property. Therefore, misusing the constraints can cause significant errors and create unintended consequences when the ontology is classified or inferenced. Consider the example the `hasBoundary` property. The property's domain is `BodyOfLand` and range is the union of `Latitude` and `Longitude` classes. However, no error is raised when this property is used to describe the relationship between individuals of the class `Country` and the individuals of `Ocean`, even though `Ocean` is not part of the specified range. If `Brazil`, an individual of `Country`, applies the `hasBoundary` property to associate with `AtlanticOcean`, an individual of `Ocean`, the OWL statement will reads "*`Brazil hasBoundary AtlanticOcean`*." Although this relationship does not cause an error by itself, the problem occurs when the ontology is classified. Since `hasBoundary` property has defined domain and range axioms, `BodyOfLand` and union of `Latitude` and `Longitude` respectively, and since that does not align with the property applied to individuals of `Country` and `Ocean`, the classifier will make inferences based on the domain and range specification. In this scenario, the classifier will infer that `Country` is a subclass of `BodyOfLand` and `Ocean` is a subclass of `Latitude` and `Longitude`. Furthermore, if `Ocean` is defined as disjoint from `Latitude` or `Longitude` in its class definition, then OWL will raise an error because disjointed classes cannot have a superclass-subclass relationship. This unintended result of class subsumption is an error that can be avoided if the developers fully understand the consequences of designating the property domain and range. For most developers, it is recommended that they do not specify the domain and range, reducing the chances of serious errors in the ontology.

d. Primitive and Defined Classes

Unlike other languages, OWL differentiates between "primitive" and "defined" classes. Primitive classes, also referred to as "partial classes," are those defined only by *necessary* conditions or restrictions. Defined, or "complete" classes,

[10] Axioms are general statements or assumptions accepted as true without demonstrated proof.

have at least one *necessary and sufficient* condition. The difference between a primitive and defined class is the level of completeness associated with the class definition. Reasoning tools can base their classification inferences only on defined or complete classes; no definitive conclusions can be made on primitive classes.

In the Geography ontology, `CoastalCountry` is a defined class because it contains necessary and sufficient conditions as part of the class specification as shown in Figure 56.

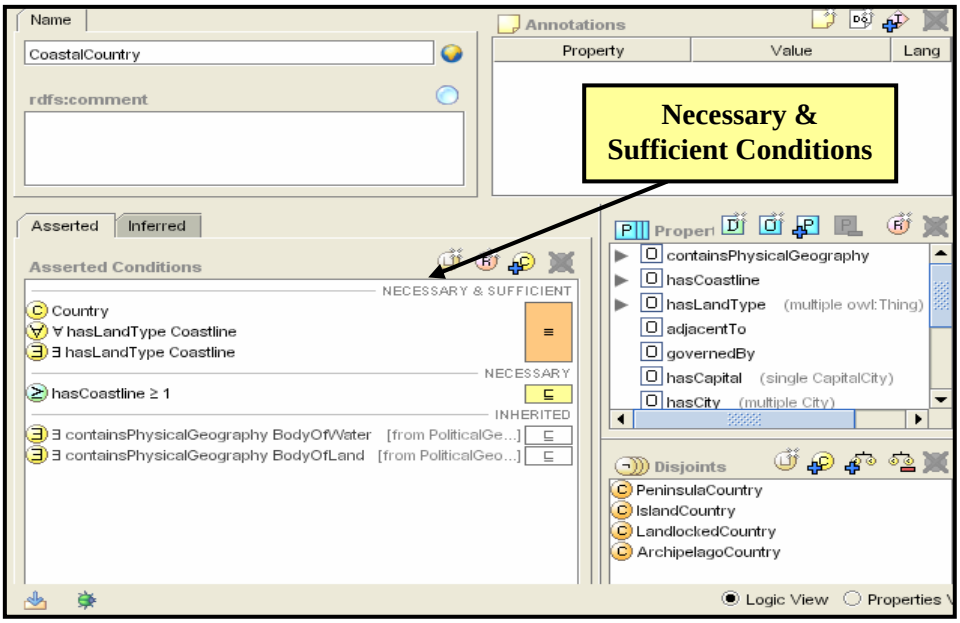

Figure 56. Defined Class Example

The necessary and sufficient conditions of the `CoastalCountry` class imply that *any* class that is country and has a land type of `Coastline`, amongst other things, is, by definition, a `CoastalCountry`. If this class was defined as primitive, with necessary conditions only, such unambiguous inference cannot be made. The primitive restrictions are insufficient to infer that the satisfaction of these conditions implies that it is the named class. It is crucial for developers to understand that unless classes are complete, using necessary and sufficient conditions, the classifier does not attempt to inference class subsumption. While the primitive class can only define its conditions, defined class are also defined by them. This difference is shown in Figure 57.

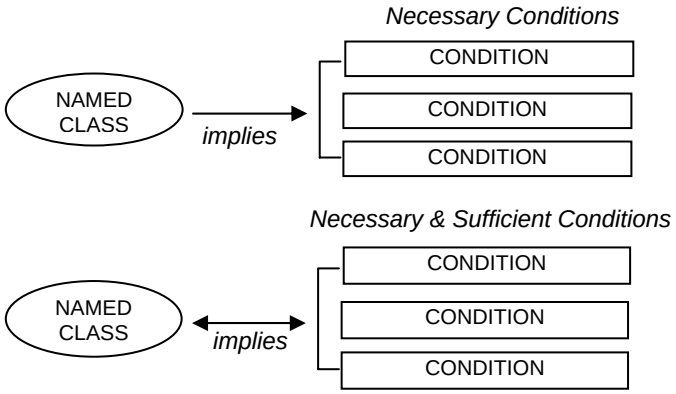

Figure 57. Necessary *vs.* Necessary & Sufficient Conditions

The OWL construct used to indicate the defined class condition is `owl:equivalentClass`. As mentioned in the previous chapter, this construct states that the class being defined has the same description, or list of individual members, as the conditions specified under the `owl:equivalentClass` tag. The necessary and sufficient definition of `CoastalCountry` using `owl:equivalentClass` is shown in Figure 58.

```
<owl:Class rdf:ID="CoastalCountry">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasLandType"/>
          </owl:onProperty>
          <owl:allValuesFrom rdf:resource="#Coastline"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#hasLandType"/>
          </owl:onProperty>
          <owl:someValuesFrom rdf:resource="#Coastline"/>
        </owl:Restriction>
        <owl:Class rdf:about="#Country"/>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Figure 58. Definition of `CoastalCountry` in OWL

Classifiers cannot make assumptions about class subsumptions without the "necessary and sufficient" descriptions that makes classes complete. OWL ontology experts argue that developers do not sufficiently understand the importance of defined classes and wherever applicable, many classes should have a complete definition [Horridge, 2004, 57].

e. Complex Classes: Proper use of Logical Operators "AND" & "OR"

Complex classes are built from joining simpler classes using logical operators such as "AND" (∩) and "OR" (∪). Complex classes are named classes, but with their restrictions stated under an anonymous class declaration. A class created using the AND (∩) operator is an *intersection class*. An intersection class combines two or more classes, using an anonymous class description that restricts the individuals to the members of the intersection of these classes. A complex class created using the OR (∪) operator is a *union class*. While the intersection class is made up of only the member belonging to *all* classes specified, a union class encompasses all the members of the classes included in the union, as diagramed in Figure 59.

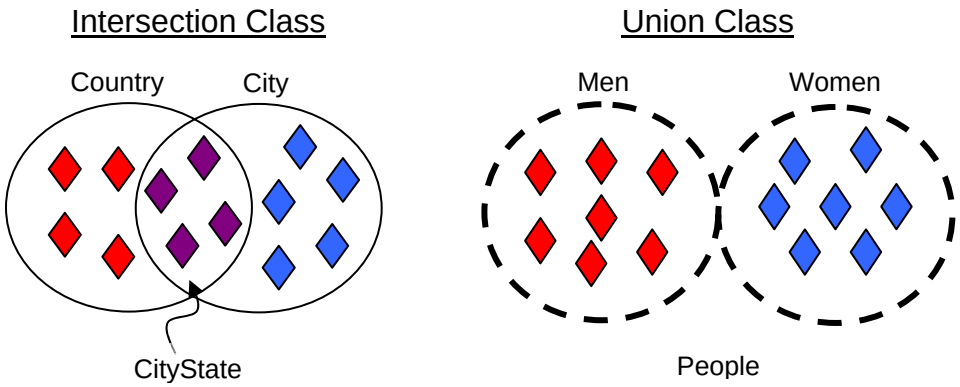

Figure 59. Intersection Class *vs.* Union Class

The application of intersection and union operators are commonly misused because the logical conjunction and disjunction do not intuitively correspond with the linguistic use of "and" and "or." The English statements such as "Name all the political

geography entities that are city *and* country," does not make it distinctly clear whether it refers to entities that are *simultaneously* city and country, or both entities that are city and entities that are country. In its logical use, AND refers to the intersection, which make up the subsection, including members that belong to every intersected class.

The Geography ontology example of an intersection class is `CityState` (`City`∩`Country`). This class is defined in a way that the members must belong to both the `City` and `Country` classes; they are simultaneously individuals of `City` and individuals of `Country`.

A union class, with the logical use of OR operator, contains all the members of each class included in the union. In the Geography ontology, the `People` class is a union class, (`Men`∩`Women`). Members of the People class include all the individuals of `Men` and all the individuals of `Women`.

6. Classify Ontology with a Reasoning Tool

One of the main advantages for developing an OWL-DL ontology is its compatibility with classification or inferencing tools. These tools validate and find new classifications of the class hierarchy based on the class descriptions. The inferred classifications provide the developers with error-checking as well as recommendations on how the classes should be organized. This is tremendously valuable, especially with a large and complex ontology, because it allows the developers to verify the consistency of the class descriptions with the overall schema of the ontology. It is recommended that after every iteration of class descriptions, the developer should invoke the classifier to check the validity of the definitions. Classification tools, such as RacerPro, output any errors or inconsistencies they find in the ontology.

Classifiers are also important for identifying multiple class inheritances. Multiple inheritance occurs when a class belongs to more than one superclass. When this happens, the inheriting class takes on the characteristics of all of its parent classes. Based on necessary and sufficient conditions of the classes, RacerPro finds classes that should be subsumed under more than one class. Although it is possible for the developers to designate multiple inheritance classes manually, it is recommended that the they create a

simple class hierarchy and let the classifier infer the multiple inheritances based on the class descriptions. It is argued that this method allows for a more manageable and modular ontology, which minimizes errors and maximized reuse of the ontology [Horridge et. al., 2004, 69].

Using Protégé as the ontology editor, the Geography ontology can be classified using RacerPro as the backend reasoning engine. Figure 60 shows the developer's ontology before classification.

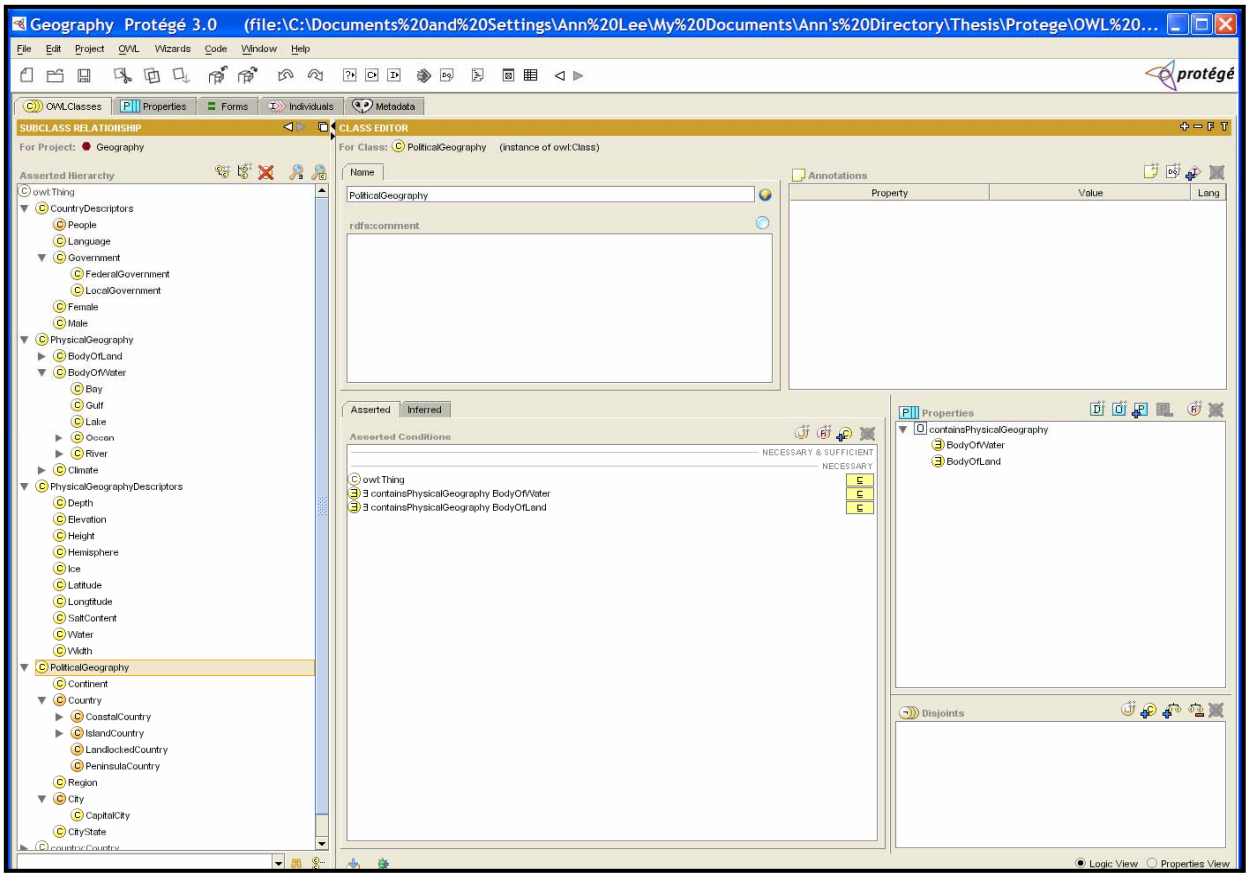

Figure 60. Ontology Before Racer Classification

The ontology, shown in Figure 60, consists of primitive and defined classes, differentiated by the yellow and orange icon colors respectively. The goal of classifying the ontology, based on the conditions and restrictions of the user-defined classes, is to find inferred relationship. This is especially important as an ontology grows in size and

complexity. However, even when an ontology is relatively simple, RacerPro finds and checks the common classification errors made by developers.

When the Geography ontology is classified, RacerPro finds the errors as shown in Figure 61.

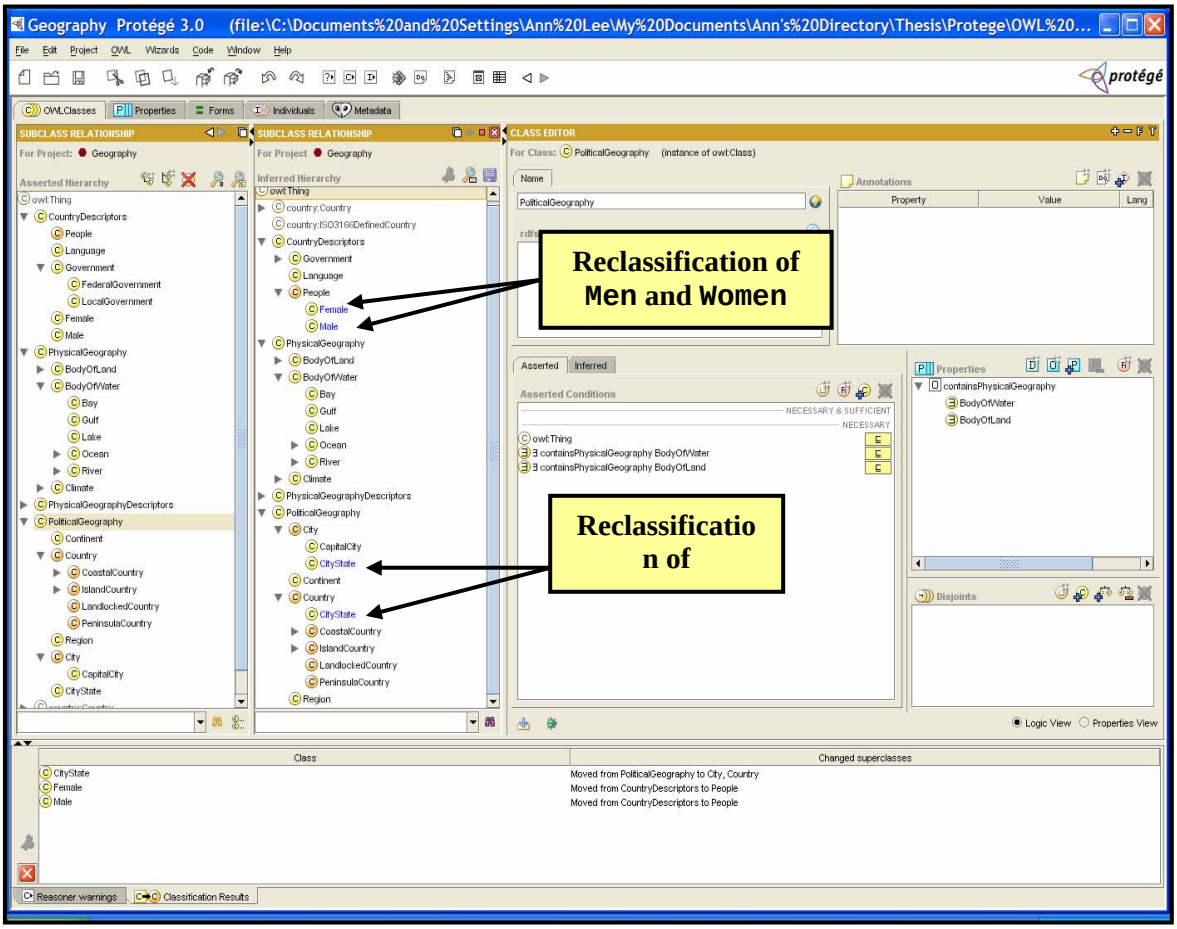

Figure 61. Ontology After Racer Classification

According to RacerPro's inferencing engine, the user-defined Geography ontology, represented under "Asserted Hierarchy," has three classification errors. First, RacerPro infers that `CityState`, which was defined above as an intersection of two classes, `City`∩`Country`, has multiple inheritances; `CityState` is a subclass of both `City` and `Country`. RacerPro infers that the since the complex class description, the intersection of two classes, is defined as a necessary and sufficient condition, is should be

subsumed under `City` *and* `Country` classes separately. RacerPro's second and third inference results state that the `Men` and `Women` classes should be subsumed under, rather than being siblings of, the `People` class. Like the `CityState` example, the class description of `People` is complete or defined, implying that the satisfaction of the class conditions infers equivalence with the class itself. Since the `People` class is defined as a union class, `Men`∩`Women`, all the individuals of these two classes also belong to the `People` class.

7. Create Individuals and Fill Property Values

OWL instantiates the ontology classes by creating individuals. Individuals represent the actual real-world entities of the interested domain that the ontology is attempting to categorize and link by property relationships. Furthermore, as shown throughout this chapter, individuals are used as part of class description and restrictions. As stated in Chapter Two, there are specific OWL constructs used to denote semantics of individuals, such as `owl:hasValue`, `owl:sameAs`, and `owl:differentFrom`. Likewise, individuals are used to define enumerated classes using `owl:oneOf`.

Many individuals that are included in an ontology are determined early in the development process, when the domain concepts are informally listed in Step Two of the development methodology. The concepts that were at the lowest level of specification, or cannot be grouped as a class, become the individuals. Unlike the other entities of an ontology, such as classes and properties, individuals are the actualization or instantiations of the descriptions. In the Geography ontology, some of the concepts appropriate as individuals are `Italy`, `France`, `Mexico`, `Rome`, `VaticanCity`, `PacificOcean`, `GangesRiver`, `MtVesuvius` and `LakeOntario`.

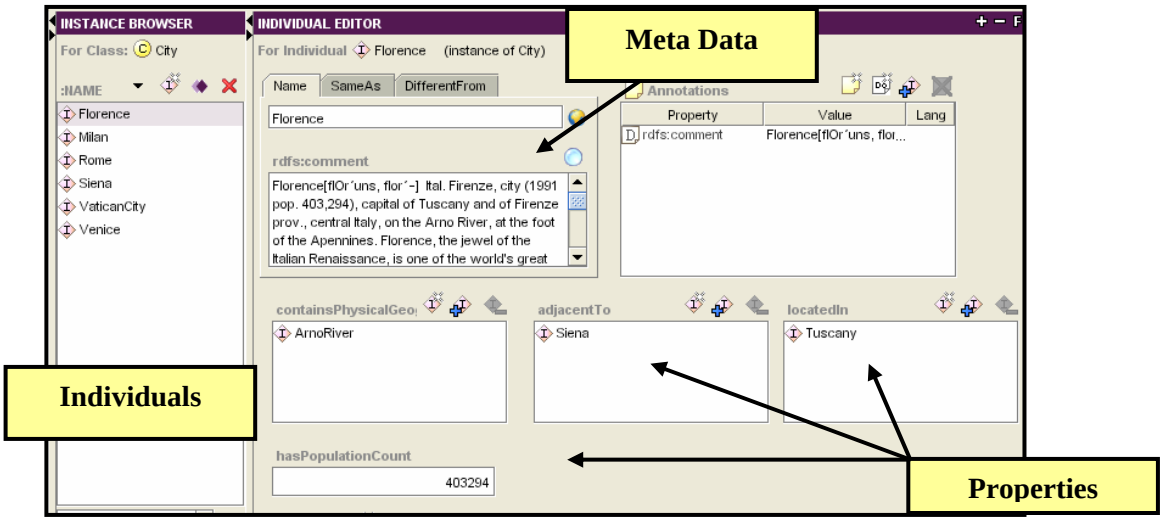

Figure 62. Example of the Individual Florence

Figure 62 shows the list individuals of the `City` class. For each individual, there is an associated list of properties specified in the class definition. These property values are determined within the individual description, as shown in the Protégé editor window. Since properties denote relationships between individuals, or between an individual and a datatype string, the developer inputs these values at the individual instantiation stage of the developments. For example, the individuals of the `City` class have the `containsPhysicalGeography`, `adjacentTo`, `locatedIn`, and `hasPopulationCount` property values to be filled as part of the individual instantiation. The instance `Florence` fills those properties slots with the appropriate values as specified in Figure 62.

Although this step is the least difficult step of development stages, it is the most time consuming. Depending on the domain and scope, the number of individuals can grow tremendously large. However, as long as the schema of the ontology is fully developed and structurally valid, managing the individuals should not pose a challenge.

E. OTHER CONSIDERATIONS FOR ONTOLOGY DEVELOPMENT

OWL allows an ontology to import other ontologies. In the famous Wine ontology, the creators import the Food ontology to describe and pair the various types of

wine with food. By importing the Food ontology, the developers are able to make use of all the Food classes, properties, individuals, and axioms as part of the Wine ontology and as part of the Wine class descriptions. Developers can also *extend* the imported ontology by adding further description of the Food classes. It is important to understand the difference between ontology reference and importing. References to other ontologies are commonly made using namespaces, such as `rdf` and `rdfs`, but the references do not allow the user to manipulate the objects of these ontologies. Importing allows the developer to have access to all of the axioms and objects of the ontology that are unavailable by reference. Furthermore, OWL imports are cyclic in that Wine ontology can import the Food ontology and the Food ontology can import the Wine ontology.

Ontology imports are integrated to the existing ontology using namespaces similar to references. The namespace is associated with the URL, where the ontology is located. The Geography ontology imports the `countries.owl` ontology, which lists the ISO 3166 country codes, available through Protégé ontology library.[11] This URL for this ontology is http://www.bpiresearch.com/BPMO/2004/03/03/cdl/Countries, which serves as the default namespace (Figure 63).

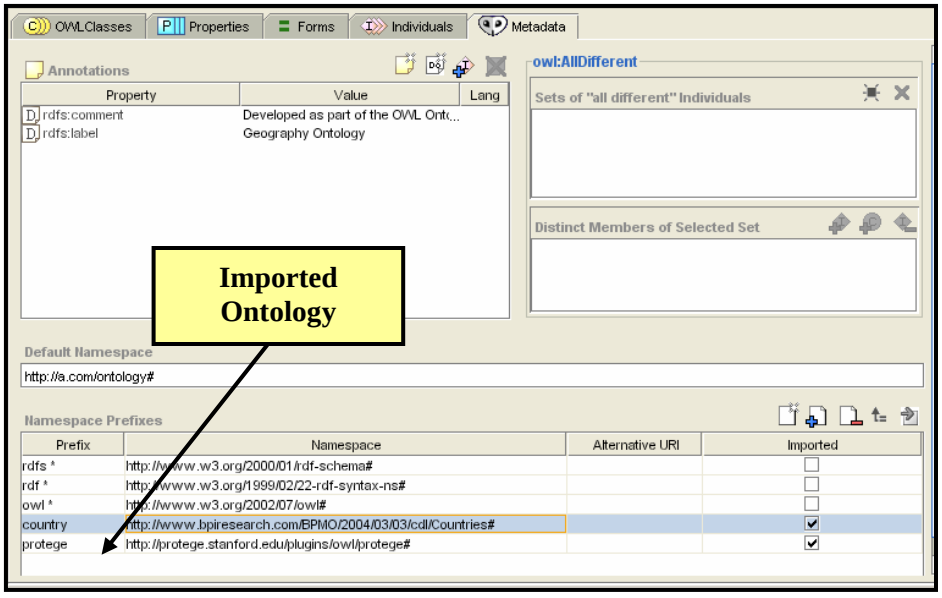

Figure 63. Importing Ontologies with Protégé

[11] Contributed by Dieter E. Jenz, Jenz & Partner GmbH, http://www.jenzundpartner.de/index.html.

The namespace URL has a # sign attached to it as a separation marker between the URL and the entities within the ontology. The namespace, which uniquely identifies the imported ontology, is associated with every entity of the imported ontology. However, rather than including the long URL with every class, property and individual, a prefix is used in its place. In this case, the prefix "country" will be used as the namespace. Once the Country ontology is imported, all the entities are “visible” to the geography ontology, as shown in Figure 64.

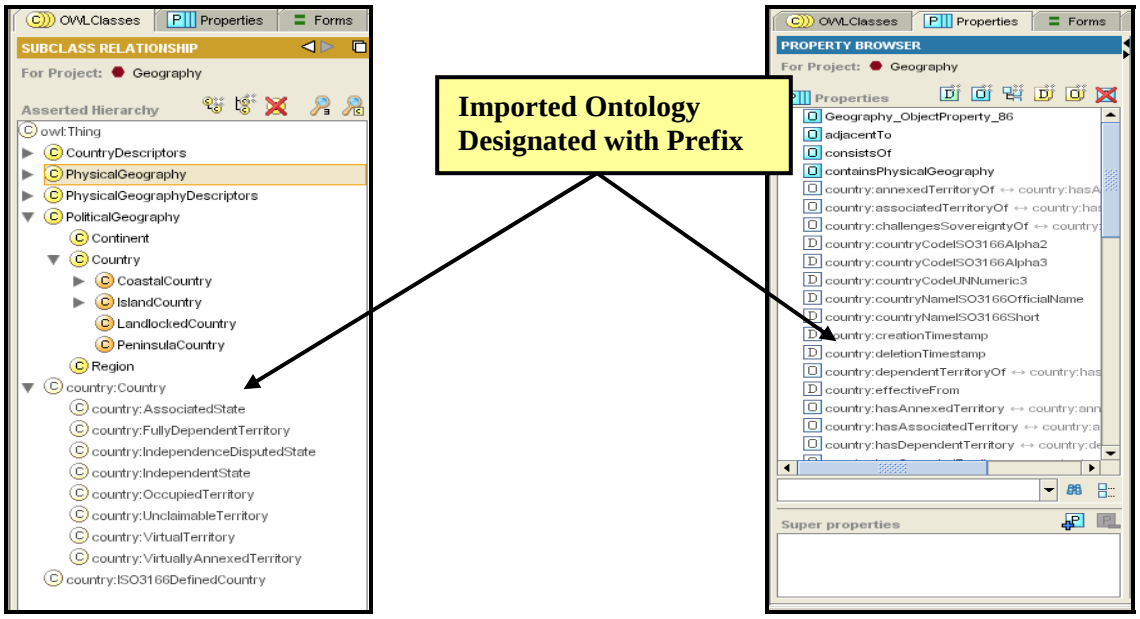

Figure 64. Classes and Properties from Imported Ontology

Figure 64 shows the imported ontology has a prefix associated with every class and property. Now these objects can seamlessly be integrated into the Geography ontology as class descriptions and even take on extensions unique to this ontology.

F. CONCLUSION

This chapter provided a methodology for developing an OWL-DL ontology. The recommended seven steps approach, described above, should be used as a guide for SMEs to build an ontology in their areas of expertise. As emphasized earlier, since the scope and application of the ontology determines the content and structure of the ontology, a significant effort should be spent on understanding the goal of the ontology, as described in step one. Once that is defined, informally listing the relevant terms of the

domain, step two, is the best method to finding the appropriate class objects of the ontology. Step three organizes those concepts into the class hierarchy. It is important at this stage to understand the difference between classes and individuals as well as the relationship of subclasses. Next, define the properties of the domain, as described in step four. OWL offers multiple property constructs. Understanding the semantics of these types of property and the kinds of relationship they imply are important in creating a rich ontology. Step five involves using these properties and other constructs to restrict and describe classes. Developers are advised to avoid common errors by using existential restriction as the default and using closure axioms to further limit the class definition. Likewise, it is important to remember that OWL is a language with open world reasoning. All necessary description should be stated explicitly. This step also explains the difference between primitive and defined classes. Classes are categorized as defined only when they use necessary and sufficient conditions as part of their descriptions. Once the classes completely defined, classification engine is used to inference the ontology, as stated in step six. It is at this stage where ontology validity and consistency are checked using an inferencing tool such as RacerPro. And finally, step seven describes how individuals are instantiated. They represent the real-world entities of the ontology's domain, rather than their abstractions.

Although these seven steps were described linearly, the development process is iterative, as in the spiral model. It is likely, and even recommended, that the developer move forwards and backwards through the steps as necessary to improve and modify the ontology. And like other systems, success of an ontology depends on good management and maintenance. The structure of an OWL ontology makes it suitable for maintenance and updates.

Given the difficulty of modeling real-world domain and knowledge into abstract ontological model, developing any ontology is a challenge. However, with a thorough understanding of OWL semantics and detailed planning of the development process, SMEs and others developers can build a useful knowledge representation system.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ONTOLOGIES AS KNOWLEDGE BASES

A. INTRODUCTION

We define an effective search as one which returns to the user a set of highly relevant results. Using the appropriate keyword(s) is essential for successful research when performing a conventional internet search. Very broad keywords will result in a large number of hits, many of them useless. In order to take advantage of currently available search engines to return topic-specific results, a highly relevant list of keywords and phrases needs to be formed. To develop such a list, users need to have access to knowledge of the domain of context. An ontology, which is model of a domain of context, can support the identification of precise and relevant keywords.

The Ontology-Aided Knowledge Discovery Assistant (OAKDA), pronounced "Oak D-A," developed as part of this thesis is an application that attempts to assist users to improve their Web searches by providing domain context to the search word or phrase. By navigating the ontology, users are assisted in finding a relevant set of key terms that will aid the search engines in narrowing, widening, or refocusing a Web search. The aim is to enhance the relevance and precision of the returned results through the use of a context provided by ontologies associated with each search. Additionally, in the process of mining the ontology, the users can discover knowledge about the concept of interest and other related terms in the domain. This chapter focuses on the purpose and motivation for the OAKDA and how ontologies can be used to augment the tools used to manage the vast amount of information and resources available in the Web.

B. MOTIVATION FOR USING ONTOLOGIES

An ontology can be defined as a formal explicit description of concepts in a domain of discourse, properties of each concept describing various features, attributes of the concept, and restrictions on these properties that are specified by semantics, or rules, that follows the “rules” of the domain of knowledge (Ushhold et al., 1996, X). For these reasons, ontologies may have use as knowledge bases (KB) for an application attempting to add context to a particular search word or phrase. By navigating the ontologies, users

can understand the context of a particular concept as well as the relationships it has with other concepts. Interest in the use of ontologies as knowledge bases is growing rapidly. If the availability of these ontologies increases, their importance may become more significant. Since no one ontology can cover all aspects of a domain and no two ontologies of the same domain will be the identical, users receive greater value by accessing as many ontologies as possible. Similarly, ontology developers can benefit from each other since ontologies can be built on top of others, expanding the breath and depth of any domain. Once developed, ontologies can be widely distributed and shared and used by both people and application systems. This scenario is what makes ontologies potentially very valuable.

The fundamental purpose of an ontology is to improve the ability of humans and machines to make judgments about data. Humans obtain and process information by reading the written word, whether on paper or on a computer screen. Our understanding of how humans decipher and process written language is not well understood and consequently, we've not managed to endow our machines with the same capabilities. Therefore, when machines require information, it must be in a language and structure that can be understood by them. This is where ontologies are valuable by providing a method of translation. Undoubtedly, there is something lost in the translation for humans. Ontologies require people a good visualization strategy to be more easily accessible. For human beings, ontologies are more limited in terms of their descriptive power as compared with prose but can be useful in certain situations to provide a succinct overview and hierarchy of a domain.

Regardless of the growing variety of applications using ontologies, the benefits of using a knowledge representation system in a form of an ontology are reusability, interoperability, reliability, maintenance, and knowledge acquisition. Although the method of making this communication differs with applications, an ontology allows both humans and machines to have a method of communicating a domain knowledge in a consistent manner.

According to Jasper et al., there are four broad categories of ontology applications, namely neutral authoring, ontology as specification, common access to information, and ontology-based search. (Jasper et al., 1999, 6)

In "neutral authoring", information is documented in a single language, which is then converted into different forms for reuse in various target systems. The main motivations for using this type of ontologies are the cost benefits of reuse and the portability of knowledge across multiple applications. In order to support neutral authoring, supporting technologies such as a unidirectional ontology translator is required.

"Ontology as specification" applications use a single ontology of a particular domain as the knowledge specification basis for developing a particular type of software. Since the software relies on the ontology to provide specific information of a domain, the ontology requires rich semantics with as little ambiguity as possible. Unlike the neutral authoring approach, this application does not translate the ontology so much as it guides the target software development. Benefits of such systems include documentation, maintenance, and reliability of the domain knowledge.

Ontologies used as "common access to information" translate information into multiple formats. Using a mapping technique, the ontology renders sharing information between different platforms intelligible by using a shared understanding set of terms. The ontology provides a way of interoperability and knowledge reuse of disparate systems. Supporting technologies include translators and parser generators.

Finally, "ontology-based search" applications are used to search information repositories for relevant resources. The motivation of using an ontology to assist the search is to retrieve a more precise result. These applications require technologies such as ontology browsers, search engine and inferencing tool.

The OAKDA falls into the last category of ontology application. The goal of this application is to assist users in obtaining better search results by exploring the knowledge contained in ontologies. Rather than relying on brute Web search engines approaches, using OAKDA to "explore" a relevant domain and all the related concepts, relationships and properties of a search term would lead to a more effective list of results. Since the

tool has the capability to traverse the pertinent ontology that relate to the user's area of interest and graphically view all the relevant concepts and their relationships, it allows the user to explore and better understand the domain of interest. This is a different method of retrieving and discovering information than a simple brute Web search.

C. KNOWLEDGE DISCOVERY USING ONTOLOGIES

It is best to illustrate the benefits of using ontologies to discover knowledge by using OAKDA with a number of example ontologies. The examples presented in the following three sections show how mining ontologies assist in domain knowledge discovery and the search for the right resources on the Web in the domains of wine, cartoon and geography.

1. The Wine Domain

Since the proliferation of the web pages on any topic imaginable, when someone wants to find information on a given subject the internet is now the first place to search. Whether it is for a profession, academic or personal purpose, the phrase "Google it" has become the ubiquitous solution. However, depending on the topic, "Googling" can actually provide more questions than answers. For example, if a user wants to search the Web on the topic of *wine*, there is no limit to the kind and number of resources produced by brute force search engines like Yahoo! and Google. For instance, a search of the word "Bordeaux" in Google results in 16.4 million hits, including sites for tourism to the Bordeaux region, the Université Bordeaux, as well as Web sites selling Bordeaux wine, as shown in Figure 65.

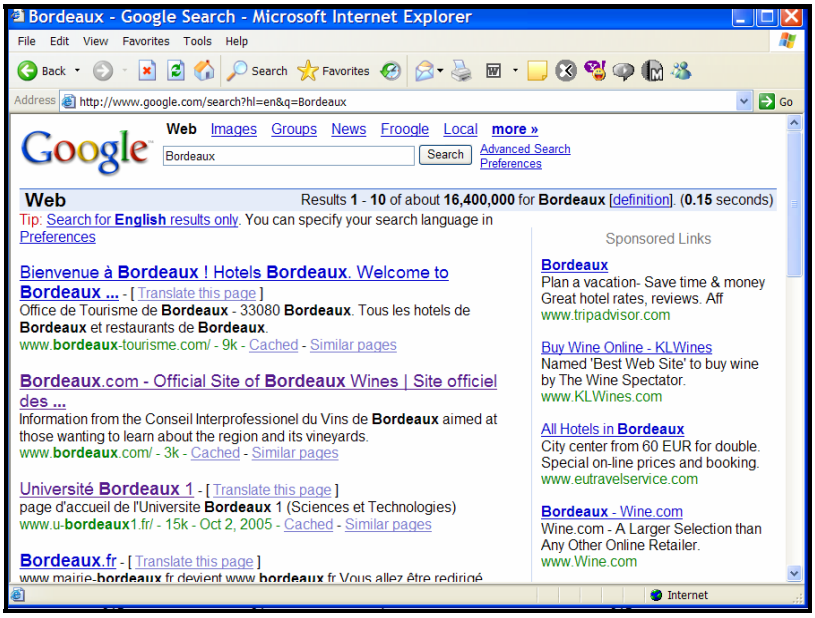

Figure 65. Google Search Results

These resources, while numerous, are not useful to someone with limited knowledge about wines in general and Bordeaux wines in particular. In fact, the information overload may creates more problems than solutions. When a user is unfamiliar with the search term, he or she greatly benefits from understanding the domain knowledge of the concept of interest. In other words, the user should learn the context in which the search term or topic belongs.

One way of gathering information is to read a book in that subject or even peruse through all the various Web sites that the search term retrieves. The first method may be the most effective way of obtaining thorough knowledge on a topic, especially if the subject involves complex ideas or relationships. The second method of perusing all the various Web sites may be helpful and by process of elimination one can deduce the appropriate context of their search. However, without prior knowledge of the domain context, blind search can lead to misinformation. In both cases, the knowledge discovery can be time consuming. The third method is the use of an ontology. In this case, wine domain would be graphically represented in terms of classes, instances, and property relationships.

Although humans are trained to glean information primarily from text, graphical representation can aid understanding. By representing ontology graphically, individuals may be able to "learn" certain important facts about a knowledge domain more efficiently than reading text off of a page.

In the case of the user searching on the topic Bordeaux, the user should first learn that it is a type of wine that is a special variety due to the location of its origin. This information is available if the user has a method of navigating a *wine* ontology to find how Bordeaux wine relates to other types of wine and what characteristics of Bordeaux distinguishes it from different wines, as well as other relevant concepts and relationships of the wine domain.

There are various methods to "read" or mine an ontology. It can be displayed in the OWL syntax or other ontology languages. Otherwise, for easier visualization, ontologies can be viewed using an editor such as Protégé. Figure 66 shows the Bordeaux as a class in the wine ontology in the Protégé ontology editor.

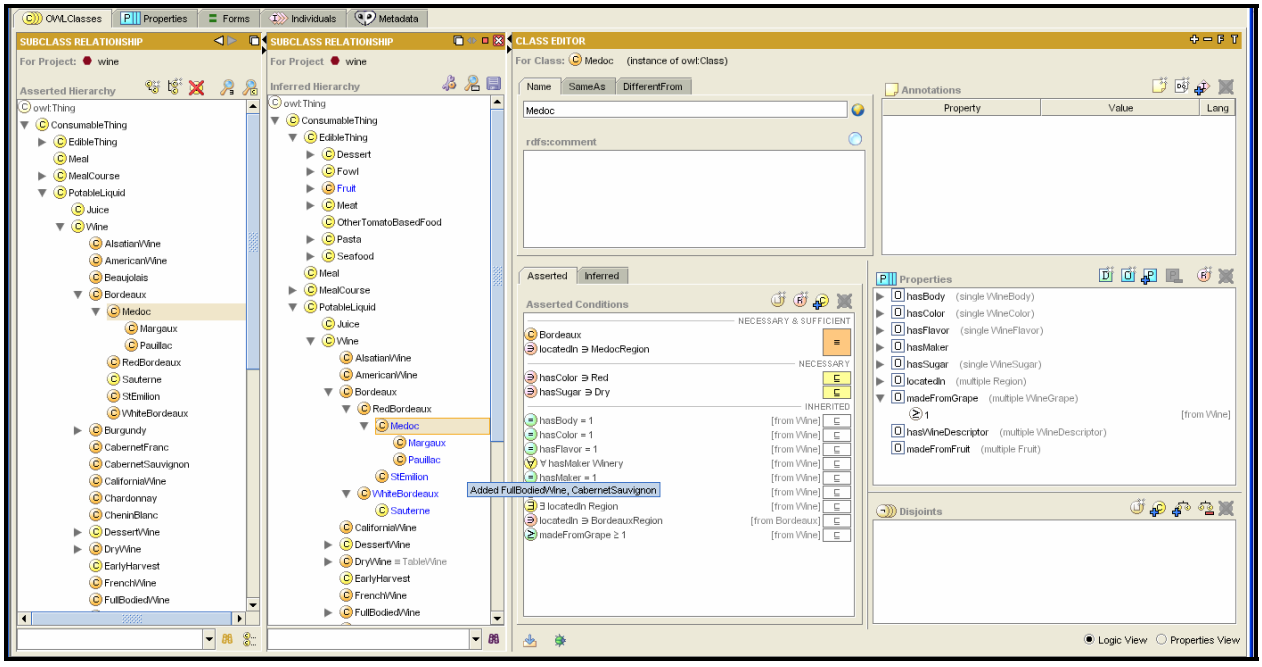

Figure 66. Protégé View of the Wine OWL Ontology

Using Protégé, the user is able to view all the classes, properties and individuals of the ontology. The asserted ontology, shown in the leftmost window of Figure 66, is

classified using the classifier RacerPro to produce the inferred ontology in the second window. Based on the asserted description, the inferred ontology shows that the Bordeaux has two major subclasses, namely Red Bordeaux and White Bordeaux. Within these two classes of Bordeaux wine, there are other subclasses, each distinguished by their unique characteristics, such as region and type of grape used, while inheriting all the traits of the parent classes, Red Bordeaux or White Bordeaux.

Protégé, like other ontology editors, makes it easy to read and navigate an ontology. However, in order to mine an ontology using an editor, the user must know exactly what ontology he or she needs as well as have access to them to load into the application. If users have no knowledge about their domains of interest, it is unlikely that they will have access to the appropriate ontologies. For those users, Protégé is not useful as an ontology-based search application. Protégé is an application more appropriate for ontologies developers rather than the users.

The OAKDA proposes to fill the gap between available ontologies representing various domain knowledge and the resources on the Web. It is an application that allows users to search its database of ontologies for their search term of interest and find the appropriate ontology that fits the appropriate domain. When users specify a search term of interest, OAKDA allows them choose from various ontologies and navigate along the most relevant ontology tree to discover related concepts and relationships that were previously unknown to them. Using the example above, one can search the OAKDA database of ontologies for the term “Bordeaux” as shown in Figure 67.

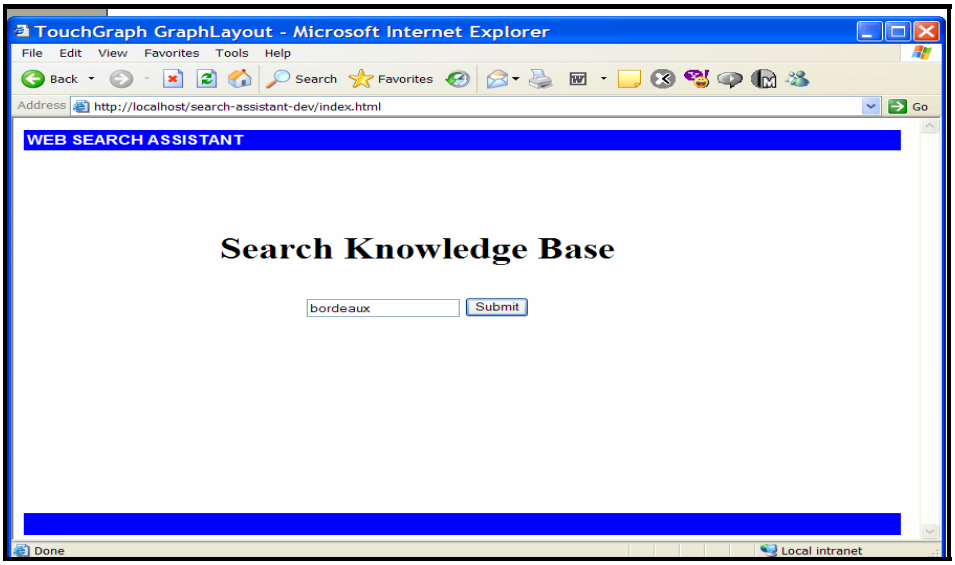

Figure 67. OAKDA Search Screen

Once the search term is submitted and OAKDA finds a match in the database, it returns a list of hits, as in Figure 68.

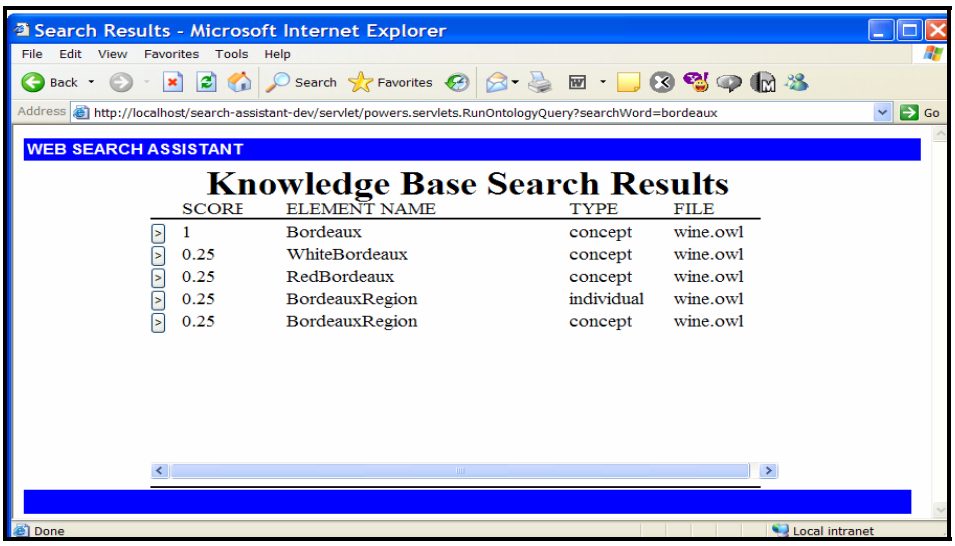

Figure 68. List of Knowledge Base Search Results

As Figure 68 shows, all the related terms are extracted from the wine OWL ontology. The "SCORE" of the search results is the ranking of match "closeness" and the "TYPE" refers to the element's OWL ontology object types, such as class, property or individual. Once the user reviews the results, he/she may begin to have a better idea about what additional information is relevant to their original search. In this case, the

user may be interested in learning more about Red Bordeaux. If the user clicks on that term, OAKDA generates a graphical representation of the Red Bordeaux class and all the objects related to this main concept of interest. Figure 69 shows the Red Bordeaux object as the middle node and all the related objects, in this case super and sub classes of Red Bordeaux Class. The direction of arrows from the Bordeaux class as well as the color of the node identifies the type of class.

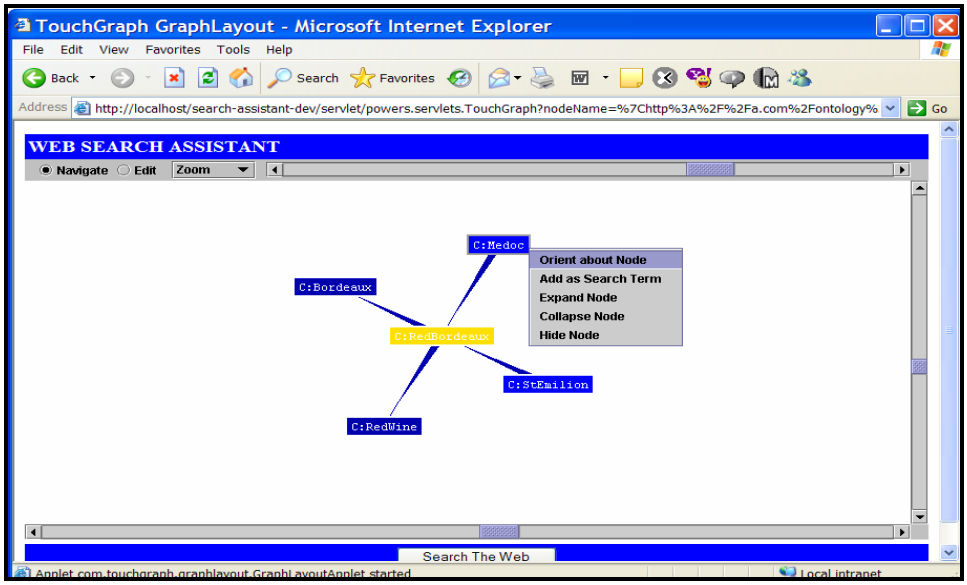

Figure 69. OAKDA View of Red Bordeaux Class

From Figure 69, the user discovers that Medoc is a subclass of Red Bordeaux for which the user would like to obtain additional information. The user can reorient the graph, using the "Orient about Node" command, which will show the Medoc class as the new central object of the graph. Figure 70 shows the result of the new orientation. Using this tool, one can easily navigate along the tree of the ontology hierarchy and view the classes, properties and individuals related to the search term of interest.

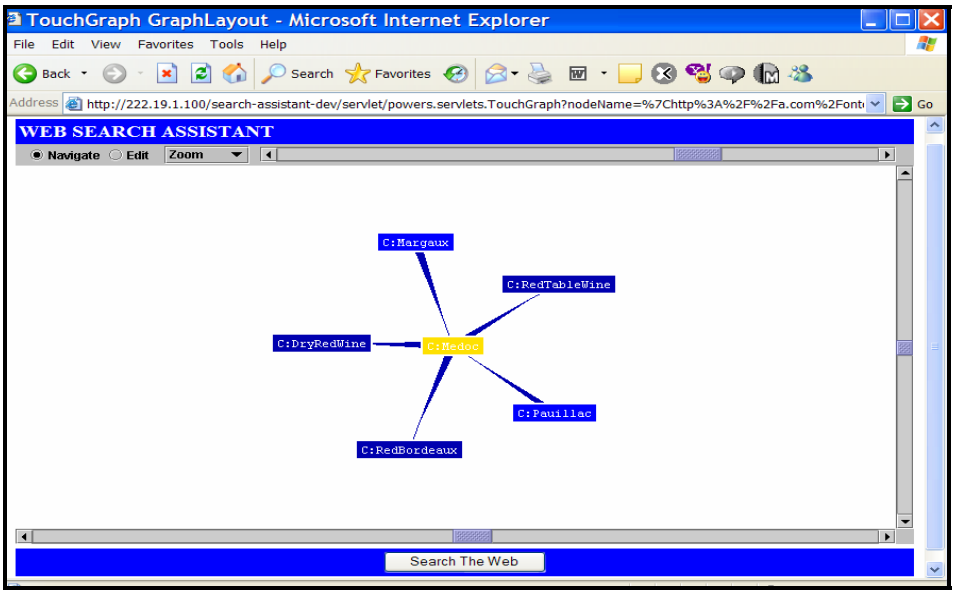

Figure 70. OAKDA View of Medoc Class

After the graph is reoriented around the Medoc class, the user finds that there are two types of Medoc Red Bordeaux wines, known as Margaux and Pauillac, which are subclasses of the Medoc class. It also shows that the Medoc class inherits its properties from three parent-classes, Dry Red Wine, Red Table Wine and Red Bordeaux. That is, Medoc is a type of Red Bordeaux as well as a red table wine and a red dry wine. Hence, the user acquires knowledge about the domain he or she is interested in by traversing the ontology that graphically displays all the immediately related concepts and relationships of the search term.

Furthermore, the OAKDA application automatically inferences any ontology loaded into the system, hiding the detail between asserted and inferred relationships between classes, as displayed in the Protégé ontology editor in Figure 66. It is the inferred ontology, classified through RacerPro that identified all the multiple inheritances of the Medoc class. When the user searches the ontologies using OAKDA, the inferencing automatically occurs behind the scene and the user is able to view a valid and accurately classified ontology.

In this simple example, OAKDA shows how it can assist users to search for additional information around the initial search term without the user having an accurate understanding of the domain or context of the term. All the concepts in this example

were in the class level of the ontology. However, users may also use OAKDA to find instances and attributes of the search term. Examples of discovering these concepts and relationships in OAKDA are explained in next two sections.

2. The Cartoon Domain

When applicable, OAKDA also displays individuals and properties, along with class concepts. For example, suppose a user is interested in finding resources on the search term "Millicent." This concept does not provide enough information to retrieve meaningful search results without additional context. To add context, also suppose that the user knew that Millicent is a cartoon character. However, even with this additional contextual information, performing a brute search on the terms "Millicent and Cartoon" lists results that does not provide user with a consistent set of contextual or domain knowledge. Instead, it would be helpful if the user can navigate the cartoon ontology to understanding exactly where Millicent fits in the world of cartoon characters.

Once the user searches the term in OAKDA, it will list all the relevant ontologies that have Millicent as a match. When the search returns the results of the match, Millicent is found as an individual in the cartoon ontology. Figure 71 shows Millicent, as the center node and highlighted in yellow, as an instance of the Mickey Mouse class. It also shows that it has property relationships with other individuals. For instance, Millicent has an “is niece of” relationship with Minnie, which in turn has an inverse property, “has niece” relationship, with Millicent. As mentioned in the previous two chapters, class properties denote relationships between individuals. Therefore, properties are not visible to the users unless there are instantiated individuals in the ontology.

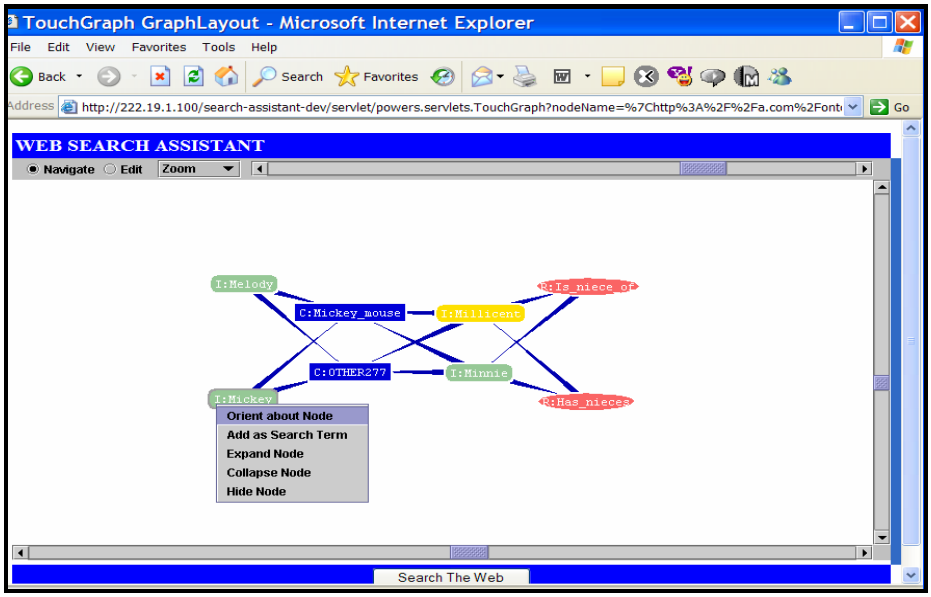

Figure 71. OAKDA View of Millicent Individual

Similar to the wine ontology example above, the best way to navigate an ontology or explore the domain using the OAKA application is to reorient the graph around different objects of the ontology graph. After understanding all the relationships around Millicent, the user may want to discover more information about Mickey. This completely reorients the ontology graph with the new object, Mickey, as the center of the graph, as shown in Figure 72.

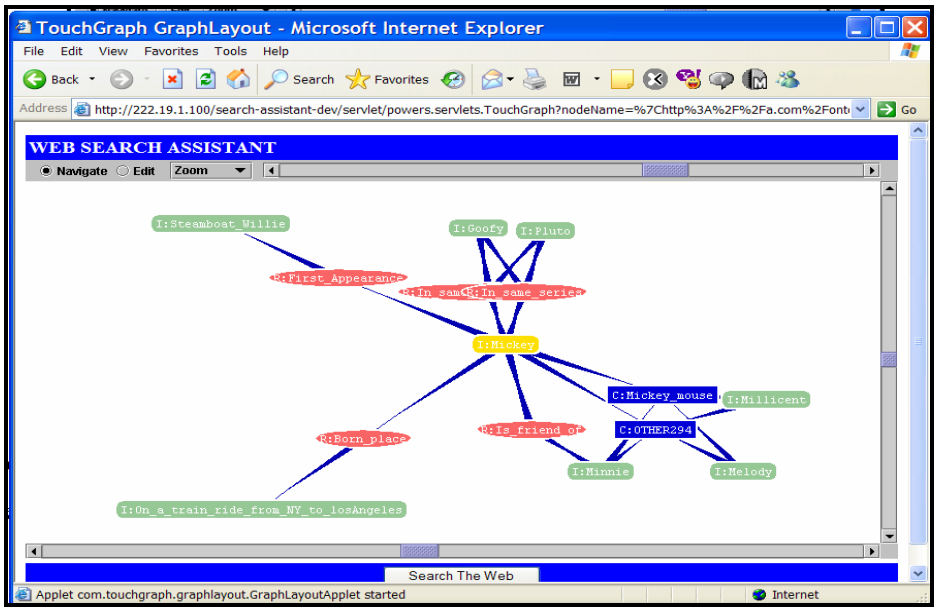

Figure 72. OAKDA View of Mickey Individual

In Figure 72, the Mickey individual is now the central point of the graph and all the immediate concepts and relationships are depicted around the new object. The new graph now shows a different section of the ontology revealing other concepts that were not included as part of the Millicent graph. When the OAKDA graph is centered on an individual, as in Millicent or Mickey, the user sees all the different relationships that the key concept has with other objects or values. Again, the user not only discovers information about the original search term, the user also gains knowledge on the tangential concepts of the domain.

Once the user obtains sufficient knowledge about the domain and the relevant contextual terms, he or she will have a better understanding of the types of information or resources to search for on the Web. OAKDA makes it easy for the user to design a list of search terms based on the graphical ontology representation. When the user finds a term that belongs as part of the search string, he or she can simply uses a right-click drop-down menu of the mouse to add it to a list. Figure 73 shows how a user adds the name of an object to the search list, using the "Add as Search Term" command.

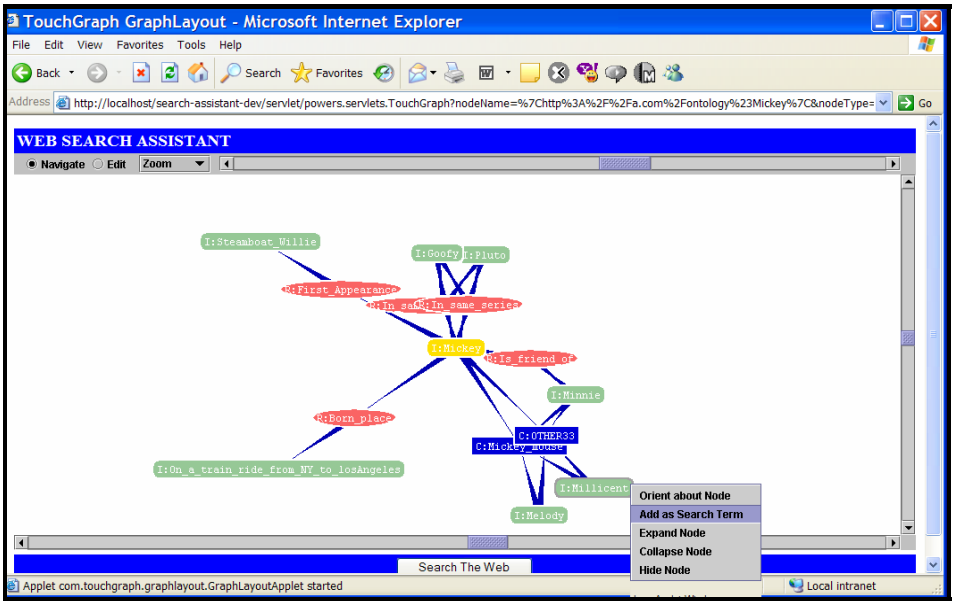

Figure 73. Adding Terms to the Search List

The user may add as many terms as desired; however, just as having too few selections can lead to an overload of Web page "hits," too many search terms may eliminate many relevant resources. It is recommended that the user choose different combinations of relevant terms to compare the results of the matched resources.

Once all the search terms are added to the OAKDA list, the user can manually edit them as appropriate. In the case of Millicent, the user has learned by using OAKDA that it is an instance of Mickey Mouse cartoon character. Therefore, the list of search terms include "Millicent" and "Mickey Mouse." In Figure 74, the user can combine these two terms using different logical operators to perform the Web search. These operators have the same restrictions and conditions as those available in most search engines. The default, likewise, is the "AND" operator as is in other search tools.

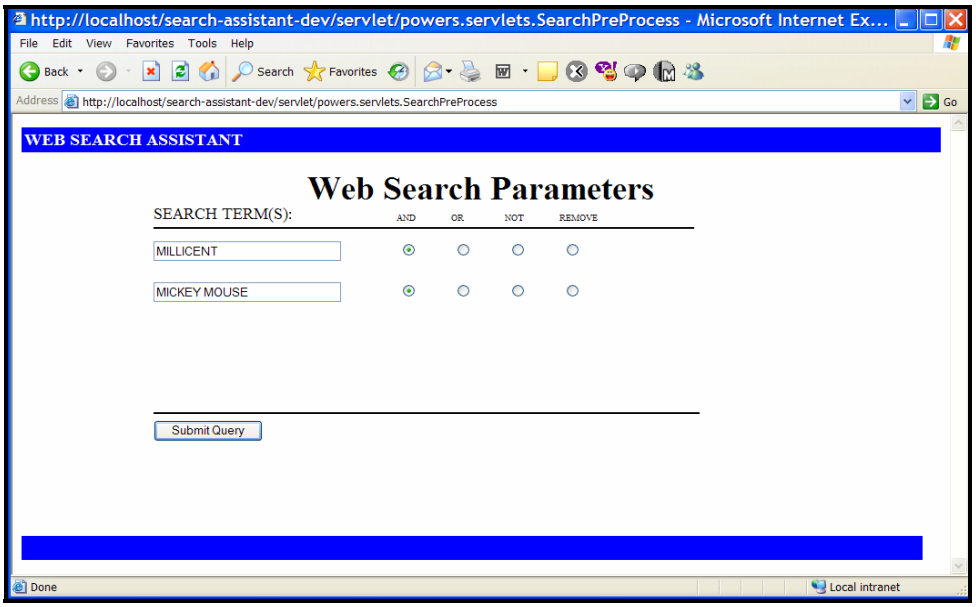

Figure 74. OAKDA Web Search Parameter List

Although the list in Figure 74 is relatively simple, it adds a context to the term Millicent that the user did not know before using OAKDA. Rather than just searching on "Millicent" or "Millicent and cartoon," the above list of terms provides the relevant context to the user's Web search. Once the list of search terms in complete, the user submits the query to OAKDA. Figure 75 shows the results of this query.

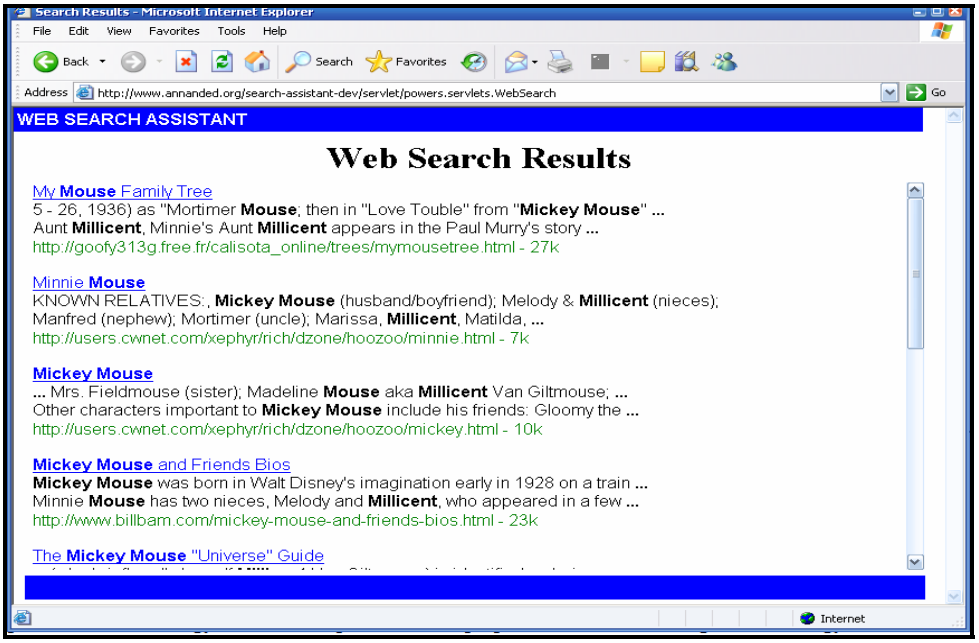

Figure 75. Web Search Results

The list of results shown in Figure 75 is derived using the same brute force search algorithm used by most search engines such as Google and Yahoo! However, by tailoring the list of search terms after discovering relevant domain or contextual knowledge of the user's initial query, the Web search matches will be significantly different from the original set. The contribution that OAKDA makes is not to change the way Web searches are performed, but to assist users to get the most appropriate information by providing a method of obtain additional knoweldge about their term of interest.

3. The Geography Domain

Even in a domain that a user may be familiar with, it is possible to discover new relationships by mining the domain ontology. If an ontology is developed as a domain knowledge representation (KR) for general application purposes, it can be used as a reference to easily learn new information. The geography domain falls under this category. As detailed in Chapter 3, this ontology was developed for the purpose of demonstrating the ontology development methodology, and also to be used as one of the ontologies in the OAKDA knowledge base. In this section, an example of how to use the geography ontology for knowledge discovery will be shown using OAKDA.

Suppose a user is deciding to go on vacation and want to find a place that meets a certain set of criteria, such as physical geography features and proximity to other locations. Of course, he or she can go to travel Web resources to discover different destination options. However, obtaining a better understanding of the world geography domain can help narrow down the traveler's options. For instance, a particular traveler is a nature-lover who is interested in finding a location that is near a lake, river, and mountain, and possibly all this one a small island. This traveler is also interested in seeing different types of physical geography in a relatively small area or a place where he can walk through a rainforest as well as a desert. In order to discover such a location, the traveler needs to find information on the different geographies of potential destinations.

Using the OAKDA application, the user decides to search on the term "island country" as a start. A list of ontology matches is presented and the user selects the most relevant concept from the geography ontology, having the highest score ranking.

OAKDA graphs the Island Country class, as a center node, and its related concepts as shown in Figure 76.

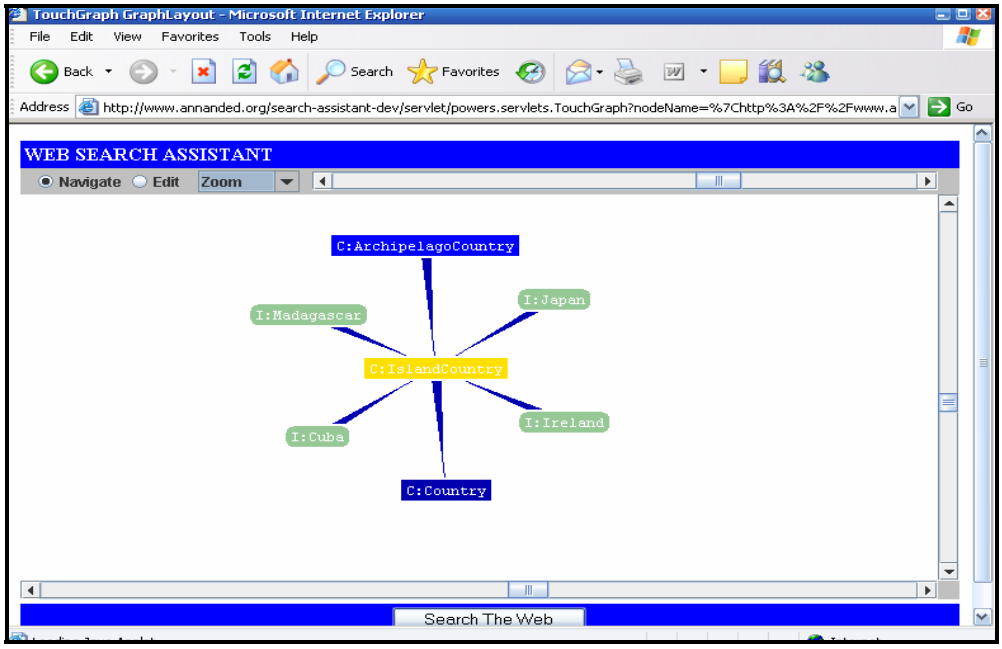

Figure 76. Island Country from the Geography Ontology

The graph shows that there are several individuals, shown as oval nodes with the arrow point towards the center, of the Island Country class. This implies that these are real-world instantiations of the island country class, and shown in the results the user confirms that these individuals are island countries. The user can learn more about any one of these individuals by reorienting the graph around that node. In this case, the traveler wants to find more information on Madagascar and reorients the graph around that individual node. Figure 77 shows the resulting graph. In general, an ontology's richness is represented at the instantiation level. While the ontology graph at the class level merely shows the taxonomy of classes and their instances, orienting the graph around an individual node provides information on its class as well as all the property relationships it has with other individuals. In other words, the information presented at this level displays all the semantic relationships the individuals have with others in the domain. Reorienting the ontology graph around Madagascar individual, the user now can see all of its relationship to its class as well as the types of relationships or links, shown

as a circle node between two individuals, it has with all the other individuals in the ontology.

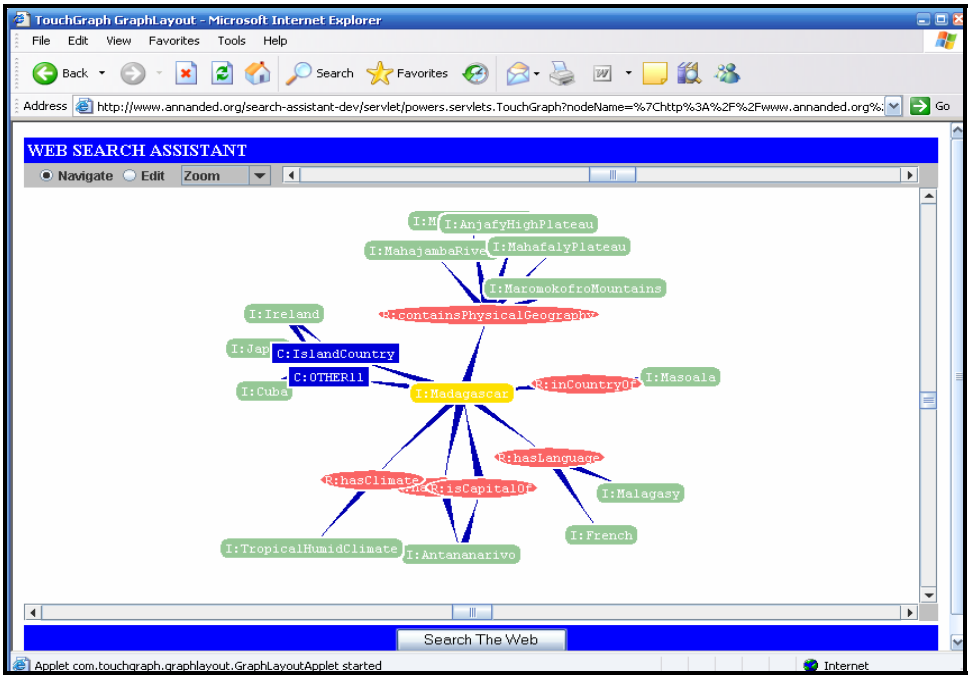

Figure 77. Madagascar Individual Centered View

According to the geography ontology, shown in Figure 77, Madagascar is an island country located in the continent of Africa and is surrounded by the Indian Ocean. It has a Tropical Humid climate and it contains such geographical features as Maromokofro Mountains, Anjafy Plateau, Mahajamba River and Lake Alaotra. Madagascar also has a rainforest, named Masoala. All this information can be gleaned from the OAKDA graph representation of the Madagascar individual. Furthermore, the traveler also finds out information he was not specifically searching for but may find interesting or useful, such as the languages spoken in Madagascar includes French, and that it is governed by a republic government. In this example, the user finds a match to his search for an ideal location for a vacation by mining the geography ontology, and can narrow his Web search for a travel agency or resource accordingly.

In this example, the user was not necessarily looking for an appropriate context or domain of a search term, rather the user used the application to find a match to a set of criteria. That is, starting from a high level concept, island country, OAKDA helped the user drill down the ontology to the representation of real-world entities, individuals, where the user can learn all the attributes or properties they have with other entities. For instance, by mining the individuals of the island class, the traveler narrows down his options to those that meet his criteria. Of course, the user could have begun his search with a different term, such as rainforest or tropical humid climate and come to the same results. No matter which node of the ontology the search or mining starts, OAKDA allows the users to traverse up and down the tree to find the related information. However, it is only at the lowest level of detail, or instantiation of classes, that one is able to see the rich semantic relationships of individuals and where find detailed and complex knowledge about the concept of interest and the domain overall.

Using OAKDA, users can focus on a narrow area of interest from relatively a large domain, such as a geography domain. Rather than search through volumes of mostly irrelevant web pages to find the right combination of information, one can quickly get to the necessary information in one site by searching for the right node and then mining its related concepts. In this example, the user understands the basic relationship of the geography concepts, but needed to know the specific instances that had the right combination of search criteria. OAKDA provided the traveler with the appropriate geography domain content information that allowed him to discover new knowledge that is of interest to him.

In the three examples of using OAKDA to assist in Web searches suggest there may be utility in using ontology to discover knowledge of value to users. It can be as simple as navigating the class hierarchy of a domain, such as the wine example or as complex as learning all the minute relationships of the individuals. The goal of the OAKDA application is to assist its users to easily discover knowledge or information that is difficult with a Web search engine alone. Furthermore, by representing the ontology graphically and reorienting around different nodes, it is easier to grasp the high and low level details than an OWL document or even an ontology editor tool like Protégé.

D. OTHER ONTOLOGY SEARCH APPLICATIONS

Using ontologies in a search capability is not unique to OAKDA. Ontology's semantic richness represented in languages that are XML based, such as RDF, OIL+DAML, or OWL, makes it a powerful tool. Specifically there are two application of interest, namely OntoSearch (http://www.ontosearch.org/) and OntoXpl, that are relevant to OAKDA.

OntoSearch is an application that searches for ontologies via the Web. The goal of this application is to encourage reuse of knowledge bases represented in ontologies and provide a mechanism for searching for existing ontologies on the Web. The user inputs the search word or words and the application finds the matching ontologies available on the Internet. The user also selects the "type" of ontology needed, namely RDF, RDFS, DAML, or OWL. Similar to OAKDA, OntoSearch also allows graphing capabilities of ontologies. OntoSearch is an important application that complements OAKDA. The success of the OAKDA depends on the availability of ontologies that covers all the various domains of the real world. It needs to build a library of ontologies as its database and the OntoSearch application is a useful tool to find those that are available on the Web.

The OntoXpl, Ontology Explorer Tool, is an application developed at the Concordia University (Canada) that assists in the exploration of ontologies using Racer as an inferencing engine of OWL DL. This application is to complement ontology editors and visualizations tools such as Protégé, with the emphasis on exploring different levels of an ontology. The user choose an ontology and the application models it into eight browsing categories, namely "file selector, "natural language" description, structural information, exploration of concept/property axioms, inspection of concept and role hierarchies, view of statistical information, inspection of A-box graph structure, and the interactive use of RACER's query language, nRQL." (Haarslev et al, 2004, 3).

Although these other ontology-based applications exist, OAKDA is unique in storing a database of ontologies and matching the user search term to the appropriate ontology. This aids the users to find relevant domain content and design Web searches

that will result in the most useful list of resources. Unlike OntoSearch and OntoXpl, users need not know anything about ontologies to use the application. OAKDA is an end-users' tool rather than a developer's one.

E. CONCLUSION

There may be great value in representing knowledge in a format that can be processed by machines, as well as humans. By leveraging the available semantics of RDF and OWL, ontologies can model the concepts and relationships of a real world domain that systems can "read" and inference based on the rules of logic. Since valid ontologies are often difficult to build, there must be incentives for the domain experts to construct them. Ontologies can become more powerful as more applications are developed to take advantage of their structured knowledge representation.

OAKDA is an application that uses a library of ontologies to look for the user search terms. It hides the details of the OWL constructs and presents the users with an interactive graph that helps them discover information that they did not previously know but might be useful for them. Its purpose is to add relevant context to user Web searches to assist in retrieving the most relevant Web page when using brute force search engines such as Yahoo! and Google.

THIS PAGE INTENTIONALLY LEFT BLANK

V. ARCHITECTURE OF ONTOLOGY AIDED KNOWLEDGE DISCOVERY ASSISTANT (OAKDA) APPLICATION

A. INTRODUCTION

This chapter describes in detail the software and hardware architecture of the Ontology Aided Knowledge Discovery Application (OAKDA) Prototype. As discussed in the previous chapters, the objective of OAKDA is to leverage OWL-DL ontologies to help end users to improve upon the formulation of their web search query by providing a framework and environment to discover terminology that makes their web searches more relevant and precise.

OAKDA enables the user to augment an initial set of search content with data derived from ontology files in one or more domains of interest. With additional search terms and better understanding of the domain of interest, users can create queries that are both focused and relevant. Results are delivered back to the users through a web search portal.

The application facilitates the above scenario by providing a framework to:

1. Pre-search a database of ontology files with the user's initial set of search terms to help the user locate the ontologies that are relevant.
2. Perform Description Logics (DL) inferencing to represent the selected ontology in its fullest meaning.
3. Provide a means to for the user to explore a relevant ontology by implementing a graphical interface that makes navigation between linked ontology elements easy and intuitive.
4. Assist the user in formatting the gathered search content into a web search query.

One way OAKDA can be used to explore an ontology is to move up and down the taxonomy of its classes. Alternatively, the application can traverse an ontology through relationships between the instantiations of different classes. Asserted content in ontologies are connected by links between instantiated classes known as "Individuals". Individuals in an ontology are related through individual/property relationships. Class definitions contain members called properties that serve to describe relationships to other individuals or concrete data types. For example, the class Human may have a property called **hasChild**. The **hasChild** property specifies that it will be "filled" by a class called

Human. Suppose we instantiate two individuals from the **Human** class and call them **Bill** and **Mary**. If **Mary** is **Bill's** daughter, then we can relate these two individuals together through the **hasChild** property relationship where **Bill.hasChild → Mary**.

The following is a typical use case for OAKDA:

- The user starts with an initial search term related to the domain of interest. The user is assumed to have little acquaintance of the knowledge domain.
- The user searches all stored OWL-DL data for string matches.
- The user navigates a chosen OWL-DL ontology to extract related information to expand the initial search term.
- The user searches the web with the new terms discovered from the OWL-DL ontology.

Because the content of an ontology is connected in meaningful ways, the end user may discover information that they otherwise might not be aware of if they have a means to traverse terminological (class) and asserted (individual, property) relationships.

An end user may also be able to uncover knowledge domains that contain their initial search term but are not contextually correct. Knowing about these knowledge domains may also useful for web search because the information can be used to signal the search engine to avoid retrieving pages that contain terminology from a domain that is not relevant.

The following sections describe in detail the system elements of OAKDA. Section B describes in generic terms, the software architecture framework that OAKDA development was based upon. Section C provides a description of each component, its attributes and roles in the application. Section D shows in detail how these components interact with each other. The conclusion provides some thoughts and lessons learned during the process of developing the prototype.

B. MULTI-TIER APPLICATION VS. SINGLE TIER ARCHITECTURES

A multi-tier software architecture is a type of client/server architecture whereby the logical components of the application are segregated by function into a composition of layers, known as "tiers", that divide up the components of the system.

Layers of multi-tier applications are defined by groupings of program functionality similar to encapsulation in Object Oriented Programming. Each layer represents a grouping of logic that possesses a small number of interfaces that can be used to send messages. This arrangement creates interconnections with a small number of interfaces between the tiers. Because only a few interfaces need to be broken to isolate the tiers, they are considered "loosely coupled". The goal of having loosely coupled architectures is to limit the effect that programming changes in one tier will have on other tiers. The segregation of logic promotes reuse of components and reduces development and maintenance costs to design by limiting the scope and complexity of each tier.

The multi-tier architecture contrasts sharply with the type of architecture used for mainframe computing, now known as single-tier architectures. These software applications consisted of a monolithic cluster of code where all function points were in scope from any part of the program, i.e. the designers were not constrained from having presentation logic make direct calls into the data retrieval logic, etc. The effect of this approach was a design where all parts of the program were "tightly coupled" and changes to one part of the program would have cascading effects throughout the code. The Single-Tier design increases the level of difficulty for software maintainability and ability to change to meet new or evolving requirements.

1. Presentation GUI Tier

This tier is composed of the software installed on the client side computer that displays a graphical user interface (GUI) for the application. The interface could be a general purpose program such as a web browser or a specialized program specially built to interact with a server.

2. Presentation Logic Tier

The presentation logic tier is responsible for provisioning the interface information to the client side and performing the steps to assemble the content which users will view and possibly interact with. There are three distinct types of presentation logic sub-tiers:

1. Web Tier: The web tier is descriptive of systems using HTTP for messaging between the client and server. The content on this tier consists of HTML, XML, CSS, and/or JavaScript that is rendered by the client web browser.

Programs on this tier are responsible for assembling content and getting the data from the business and data access tiers. The Web Tier also consists of the Web Server and Application Servers responsible to listen for HTTP request messages from multiple clients and for providing a dispatching interface to the programs that assemble HTTP responses. All data processing for this tier occurs on the server side.

2. Proxy Tier: Programs involving the "proxy" tier support a distributed computing architecture. This tier's functionality is most often provided by a third party's Web Services, described in more detail below.
3. Client Interface: Unlike the Web and Proxy tier, this component will execute on the client side of the system but its business rules are downloaded from the server. It is responsible for rendering custom displays for the end user.

3. Business Logic Tier

This tier contains the "business rules" used to perform calculations or transform and manipulate data.

4. Data Access Tier

This tier includes any components that are used to interact with or access information on the Data Tier. Examples of these components are operating system API's used to store data on host file systems or DBMS (Database Management System) access API's such as ODBC (Open Database Connectivity) and JDBC (Java™ Database Connectivity).

5. Data Tier:

Data that needs to be "remembered" by the application or is used as a driver for future processing is stored on this Tier. It is the layer that stores all the application data and is an essential part of any Multi-tier application. The data usually resides in a database or file system accessible by the server.

As indicated in the previous section, the intention behind structuring software in Multi-Tier Architectures is to enhance the ability of the system to adapt to change or at least allow large sections of a project to be reused in new applications. This approach was chosen for the OAKDA prototype in order to make it more adaptable to changes during its development and lifecycle.

C. OAKDA PROTOTYPE MULIT-TIER ARCHITECURE

Table 7 below shows an overview of the OAKDA components and their associated tiers. Additional detail describing each OAKDA tier component is provided in the section below.

TIER		OAKDA COMPONENTS
Presentation GUI		Web Browser (Microsoft Internet Explorer, Mozilla Firefox, etc)
Presentation Logic	*Web*	Apache Web server Tomcat Application server Java Servlets HTML CSS JavaScript
	Proxy	Google SOAP Web Services
	Client Interface	Java TouchGraph Applet Java Http API
Business Tier		Java (match algorithm, data transformations, graph data construction) Racer
Data Access Tier		JRacer Racer JDBC
Data Tier		MySql Database OWL-DL Ontology Files

Table 7. OAKDA Multi-Tier Component Matrix

1. Presentation Tier

The Presentation Tier code runs on a Web browser. The Web browser is not actually developed as part of the application; rather it is an off-the-shelf client software that is required to interface with HTTP messages that OAKDA receives and produces. Web browsers render Web-based document content into an interactive graphical user interface. The OAKDA application assumes that a Web browser such as Internet Explorer or Mozilla FireFox is available to the end user and that it is configured to process CSS, JavaScript, and Sun Java™ Applets.

2. Presentation Logic Tier

The Presentation Logic Tier provides the information that provisions the user interface on the client side. The OAKDA application uses the following languages and components to render a GUI presentation for the end user.

a. Presentation Logic: Web Tier

The Web Tier consists of those Presentation Logic components that use Internet based languages, technologies and methodologies.

(1) HTML (Hypertext Markup Language) is a non-proprietary subset of the SGML markup language. HTML is treated as a specification by Web Browsers for rendering viewable document content and Web forms. The bulk of OAKDA screens are rendered in HTML

(2) CSS (Cascading Style Sheets) is a mechanism for adding style to HTML documents. It performs a number of presentation effects, like positioning, which serves to augment the HTML specification. CSS is ancillary to the maintainability and reusability of Web pages by separating the presentation from the content.

(3) JavaScript is a scripting language that executes on a thread provided by the Web browser's (client side) process. Web content can be made to be more interactive and dynamic using JavaScript. In OAKDA, JavaScript is used most often to implement the Web page navigation scheme.

(4) Apache Web Server – A Web server is a software program designed to deliver data, usually in the form of HTTP messages, across a TCP/IP network between client and server computers. The sequence of events for client/server communication over HTTP is usually as follows:

- The client sends an HTTP request to the server. The request consists of the IP address of the computer running the server, the IP of the computer making the request and parameters that may specify the type of content the client is requesting.
- The Server responds with an HTTP response containing the requested content, if found.

The Apache Web Server, obtained from the Apache Software Foundation, was the Web Server of choice for the OAKDA project. It is open source and

freely available software. The Apache Software Foundation is an organization founded to facilitate the development of several software projects. Since 1999, the Apache Web Server has been the most visible and popular HTTP server in use on the Internet.

(5) Apache Tomcat Application Server – A Web server's main function is serving HTML documents to a requesting client. However, when the content of those documents is dynamic, some kind of data processing must be accomplished to perform calculations or retrieve data necessary to construct the content on the fly. Application servers are the components that facilitate the interface between a HTTP message handled by the Web server and the invocation of a specific program to perform data processing. An application server can be thought of as a type of middleware that handles messaging between the Web server and the various back-end applications, such as databases or programs that implement business logic in a data processing environment.

Jakarta, a sub-project of the Apache project, facilitates Java™ based open-source projects. The Tomcat application server is one the Apache Jakarta projects utilized by OAKDA. Its distinguishing feature is that it enables a special type of Java™ class called a Servlet to perform HTTP messaging toward the Web server and it is the program entry point for the invocation of Java™ methods utilizing the server side components.

(6) Java Servlets – A servlet is a Java™ class that is built to interface with HTTP on an application server. It reads HTTP parameters, manages HTTP sessions, and handles other services such as Web site authentication. Output data is sent using built-in servlet methods that create the HTTP response messages to be transmitted to the client via the Web server.

b. Presentation Logic: Proxy Tier

The Proxy Tier is the sub-category of the Presentation Logic that deals with the use of third party components in the architecture.

(1) Google Web Search Service – Web Services are application level services which enable inter-process communications between computers across Internet or intranet network boundaries. When processes communicate, they require the ability to transmit data arguments to "call" a far process and transmit "return" messages to a calling program. Web Services provide a framework for transmission of calls and

"return" data via XML formatted request (call) and response (return) documents. The XML requests and responses are received and dispatched by Web servers, usually over HTTP. XML and Web Services act as a middleware that eliminates the need for data processing systems to interoperate directly with each other. Since the remote procedure calls are performed in a way that is platform non-specific, the programs executing the business rules do not need to be compliant with each other's software or hardware platforms. In this way, Web services solve compatibility and interoperability problems common in other types of network based inter-process communication.

There are three key components to Web services.

- Simple Object Access Protocol (SOAP) defines the XML grammar for Web services requests and responses. A SOAP envelope is the name of the XML document that is transmitted as request or response. Inside the envelope for a request are the function call names, the parameter list and all other information needed for the peer system to make a service invocation. SOAP response envelopes are similar to requests in that they are a formatted XML documents, but they contain the "return" data of the request or fault messages if the request could not be processed.
- The WSDL (Web Service Description Language) component serves to describe the specifications for the SOAP request and response document formats. Analogously, when calling a method of a linked program, the programmer needs to know the method name, the parameter list, the data types of each parameter and the return value. A program that does not properly reference a method specification will usually not compile. In the case of Web services, the components that interact are separate entities with no "awareness" of each other. The WSDL is a way for the Web services host to advertise the correct specification that should be used to construct SOAP syntax for successful interoperability.
- UDDI (Universal Discovery Description Integration) is an internet or intranet facing directory that serves to advertise Web service capabilities available for use. The UDDI can provide the basic information needed to make contact with a Web service host including the download of a specific Web service's WSDL.

(2) Google Web Search API – Google offers free Web services client software for personal, non-commercial use. After registering for the service, the Web services client is able to send a SOAP message containing a Web search query to Google's Web services host. The host dispatches the query to Google's internal mechanisms for searching its database of indexed Web content and returns a SOAP response to client. The response contains the Web page hits generated by the submitted query.

The OAKDA application uses a Web services interface to perform Google Web searches. The response SOAP message is parsed and formatted into the HTML pages of the OAKDA application. In this way the user gets an end-to-end capability when using the application. They would otherwise have to transfer to a Web search portal when they want to perform their Web searches.

c. Presentation Logic: Client Interface

The Client Interface is the sub-category of the Presentation Logic Tier that provisions a user interface which executes on the client but whose business logic is downloaded from the server.

(1) Applets are programs written in the Java™ programming language that can be embedded in an HTML page, in the same way that image files can be included. Java™ applets are executed in the client Web browsers Java™ Runtime Environment (JRE) and are subject to restrictive policies, for security reasons, that prevent the running program from accessing the local machine's file system or from making network connection to any computer other than the program's originating host. The policy is meant to prevent malicious programmers from harming the client side computer.

The OAKDA application creates a client request to download an applet from the OAKDA Web server. The applet has add-ins that allow it to communicate with the Web server and maintain HTTP sessions.

(2) TouchGraph[12], created by Alexander Shapiro in 2001, is an open-source user interface software used for information visualization and data modeling.

[12] http://www.TouchGraph.com

TouchGraph is commonly used for visualization of data points that can be represented in directed graph format. TouchGraph was extensively modified for use with the OAKDA application because it did not have a native interface for input of the data representing the directed graph.

The modifications enabled the software to accept the information regarding nodes and the connections between them. The software renders the information in a two-dimensional display interface and simulates "virtual physics" called "Spring-Layout" which serves to spread the data evenly on the presentation screen area. Nodes connected by edges will attract each other while all other nodes repel one another. At short distances, the repulsion "force" of the nodes is stronger than the attraction force of nodes connected by the edges. The result of this is to cause all nodes to self-organize in a way where they will not be concealed by other nodes. Also, nodes that are part of a strongly connected section of the graph will tend to cluster together while weakly connected parts separate.

The software also provides functionality to manipulate the graph data on the screen. The nodes can be repositioned by using "drag and drop" mouse commands, and can be rotated in the viewable area. The interface also has slider controls that change the intensity of the node repulsion force that is compressing or elongating the edge lengths in the graph. This is useful when the graph is crowded by a large amount of nodes because the expansion will increase the readability, but at the expense of the number of nodes that can fit in the viewable field. The software also has the capability to hide whole sections of the graph so it can be scaled down to a manageable size. To restore the hidden nodes, the user clicks on an indicator icon that re-expands the graph.

In the OAKDA application, TouchGraph executes on a client-side applet. It is used to render only a portion of the user selected ontology. The nodes are represented by the ontology names for class, property, and individual data. TouchGraph directional edges connect the nodal information to show the inheritance relationships and differences in color gradation further underscore direction of inheritance between classes. All classes are depicted as blue boxes while a darker shade of blue is meant to indicate that the class is a parent of the lighter shaded class node.

(3) BOT Java™ Package – The BOT Package is client-side software used for HTTP communication. The package contains programs that have the capability to manage session and authentication between Web client and server. Jeff Heaton, who created software robots called "bots" able to automatically traverse linked Web sites and perform data processing on their content, authored the package. The "bots" require the ability to manage session and authentication to successfully access Web site information.

The HTTP Java™ class contained in the BOT package performs some of the same functions as a Web browser such as Microsoft's Internet Explorer or the Mozilla Firefox. The important contribution of the BOT package to OAKDA is the management of HTTP session information. A session is the sum of all HTTP communications between a client and the Web server. The HTTP protocol on its own only manages communications where each request and response pair is independent from the all others. Hence, HTTP is considered to be a "stateless" protocol. It is important for the Web application to be able to associate the series of request/response pairs in order to correctly differentiate between the requests and responses of multiple sessions. Without this ability, a response could get associated to the wrong client, and misdirect the end user to an incorrect Web page.

Heaton's HTTP Java™ class has methods to support cookies, which are messages sent from the Web server and stored on the client's host machine. During subsequent communications, the identifier information in the cookie is sent back from the client to enable the Web server to track a particular session and the events that the session registered. Heaton's HTTP software is also used by OAKDA to enable HTTP sessions between the TouchGraph applet and the Web server.

3. Business Tier

The Business Tier in a Multi-Tier Architecture contains the component programs that perform calculations or data transformations.

a. Racer Server

Racer is a data processing server for knowledge representation systems and description logics. Racer was developed by Ralf Möller[13], Volker Haarslev[14], and Michael Wessel[15]. Knowledge representation, a field of Artificial Intelligence, focuses on the design of formalisms that are both epistemologically and computationally adequate for expressing knowledge about a particular domain. (Baader et al., 2003, xiii) Description logics is a framework for representing knowledge in a form of individuals, classes of individuals and property relationships that describe them.

Racer acts upon terminological and asserted aspects of ontology data. In OAKDA, it is architecturally situated as a middleware between ontology markup files, such as OWL-DL, and the programs that need to access or modify them. Racer provides reasoning or inferencing as a central service. The algorithms underpinning Racer's reasoning engine guarantee "correctness", "completeness" and "decidability." Correctness means that no false conclusions are drawn. Completeness implies that all correct conclusions are present and decidability means that there exists a terminating program that can complete reasoning. Reasoning allows one to implicitly infer represented knowledge from the explicit knowledge contained in the knowledge base. (Baader et al., 2003, 43) As an example, suppose it is explicitly stated that X is to the left of Y and that Z is to the left of X. A reasoning engine infers that Z is also to the left of Y by the rule of the transitive property, "to the left of". Racer also provides services to either publish to or subscribe (query) from an existing knowledge base.

Racer divides ontology content in terms of A-Box and T-Box reasoning where the A-Box is the set of asserted relationships between individuals and classes or between individuals and other individuals. The T-Box is the set of classes and properties of classes arranged in a hierarchical inheritance relationship that describe the schema of the domain. The classes contain properties that further define them. The "A" in A-box

[13] University of Hamburg, Germany

[14] Concordia University, Montréal Canada

[15] University of Hamburg, Germany

refers to "asserted" content, while the “T” in T-box refers to "terminology" and the relationships within the T-Box which resemble a schema.

The following is a list of services that Racer has available to the end user. For T-Boxes, Racer can answer the following queries:

1. Class consistency with regard to a T-Box: Is the set of objects described by a Class empty?
2. Class subsumption with regard to a T-Box: Is there a subset relationship between the set of objects described by two classes?
3. Find all inconsistent classes mentioned in a T-Box. Inconsistent classes might be the result of modeling errors.
4. Determine the parents and children of a class with regard to a T-Box: The parents of a class are the most specific class names mentioned in a T-Box which subsumes the class. The children of a class are the most general class names mentioned in a T-Box that the class subsumes.

For A-Boxes, Racer can answer the following queries:

1. Check the consistency of an A-Box with regard to a T-Box: Are the restrictions given in an A-Box with regard to a T-Box too strong, i.e., do they contradict each other? Other queries are only possible with regard to consistent A-Boxes.
2. Instance testing with regard to an A-Box and a T-Box: Is the object for which an individual stands a member of the set of objects described by a certain query class? The individual is then called an instance of the query class.
3. Instance retrieval with regard to an A-Box and a T-Box: Find all individuals from an A-Box such that the objects they stand for can be proven to be a member of a set of objects described by a certain query class.
4. Computation of the direct types of an individual with regard to an A-Box and a T-Box: Find the most specific class names from a T-Box of which a given individual is an instance.
5. Computation of the fillers of a property with reference to an individual. Check if certain concrete domains constraints are entailed by an A-Box and a T-Box use.

In the context of the OAKDA application, Racer’s primary functions are for query and inferencing of the OWL-DL data. Racer is instructed to upload the set of OWL-DL relations into its memory and deduce any extra information present through its inference functions. Once the ontology is present in Racer’s memory, OAKDA initiates queries that deliver the nodal relationships portrayed in the graphical user interface. An example of a query is “retrieve all direct subclasses of Class “X”. A formatted query in

Racer's syntax is sent to the server, where processing takes place and the answer is delivered back to the source address of the requester.

Racer's implementation language is COMMONLISP. Racer communicates via the command line or with a TCP/IP network interface. It listens for TCP/IP requests messages and issues responses containing the requested information over the same channel. It can be configured to handle multiple simultaneous users. Since no "publish" methods are used in the OAKDA system, there is no need for Racer to maintain session state for each user. Racer is accessed by OAKDA programs through the user of the Java™ JRacer API, which abstracts Racer function calls into a format callable by Java™ programs.

b. Ontology Search matching algorithm

In the OAKDA application, the end user searches a database table containing the indexed content for all OWL-DL files loaded in the system. The indexed ontology data are accessible through JDBC calls to MySql database that stores the information. Substring matches are pulled from the database using the SQL "LIKE" command. A substring match occurs when one string partially or fully matches another string. "Regular Expressions" are used in later processing to formulate a match closeness score. A Regular Expression is a pattern of characters that describes a set of strings. OAKDA uses a regular expression construct provisioned by the Java™ "String" class.

When a substring is found, a score is computed that rates the "match closeness" (MC). This works as follows: the Query String (QS) is matched against the Indexed Ontology Element (IOE) string. The string Length Difference (LD) between QS string length and IOE string length is computed. If QS matches with IOE, the MC score is calculated as follows:

```
MC := 1 - ( LD ÷ IOE string length );
```

For example, suppose QS and IOE are given as SENT and PRESENTER. Since SENT is a substring of PRESENTER, the match score will be computed as:

```
LD :=  5 = 9 – 4
IOE string length := 9
MC := 0.44 = 1 – ( 5 ÷ 9 )
```

If QS and IOE are given as PRESENT and PRESENTER the score will be 0.77, which is correctly ranked higher than the previous query. Each indexed node is given a score between 0 and 1 to rank its relevance to the match.

Users performing a search are not limited to a single search term. Since the indexed ontology element often is a phrase, more than one term can apply when searching the database. OAKDA's programs will parse an element named `TropicalHumidClimate` as `TROPICAL HUMID CLIMATE`. A ranking algorithm was designed to handle cases where the closeness of match across a phrase needed to be computed.

Consider an example where the user query string is `TROPIC HUMID` and the indexed ontology element is `TROPICAL HUMID CLIMATE`. The overall match ranking will be calculated in the following way and as shown in Table 8.

	IOE		
QS	TROPICAL	HUMID	CLIMATE
TROPIC	**0.75**	0.0	0.0
HUMID	0.0	**1.0**	0.0

Table 8. Preliminary String Match Matrix

1. The best match will be computed for each word. The score computation matrix for individual string matches is shown in Table 2.
2. The highest scores (in bold) will be used in the overall computation.
3. Any IOE term that did not have a positive score will be assigned the average score for the all terms.
   `Average= 0.58 = (0.75 + 1.0 + 0.0 ) ÷ 3`
4. The total score is the product of all scores:
   `Score = 0.44 = 0.75 * 1.0 * 0.58`

The intention behind ranking the match similar to the ranking of page hits in a Web search. The ontology content with the highest score will have the closest match to the given search terms. The closer the score is to one, the higher ranking in the displayed list. This algorithm is crude but fairly effective. Future enhancements would include methods similar to those used by Web search portals.

c. Ontology Batch Loader

Since users benefit when there is an ontology in OAKDA's repository that contains the knowledge domain of their interest, the application contains functionality that enables end users to make contributions to the database. OAKDA enables users to submit ontology data for other users. OAKDA provides a webpage that allows end users to input the URL of an OWL-DL ontology that is hosted on the Web. When the URL is provided and the end user clicks the submit button, the OWL-DL downloads to OAKDA's host server. Batch loader programs call Racer functions that read and parse the class, individual and property nodes contained in the ontology and load them into OAKDA's internal database.

4. Data Access Tier

The Data Access Tier comprises programs in the systems that are used to interface with the Data Tier.

a. RICE JRacer API

The JRacer API is a package of Java™ classes with capability to format and invoke, via TCP/IP sockets, request messages formatted as COMMONLISP Racer Server commands. The API also can listen for Racer's TCP/IP response and bind the response to Java™ objects. The methods of the API mirror those of the Racer server except that JRacer invocations can be defined in terms of Java™ syntax and use Java™ typed objects.

JRacer was developed as a part of an ontology visualization project called RICE[16] (Racer Interactive Client Environment), built by Ronald Cornet of the University of Amsterdam. In creating the JRacer interface, Cornet provides an easy to use, reusable and documented interface between Racer and Java™ programs.

b. JDBC

Java Database Connectivity (JDBC) is a Java API built into DBMS systems which allows Java programs to run SQL statements.[17] In OAKDA, Java

[16] http://www.b1g-systems.com/ronald/rice/

[17] ODBC is another well known API and is parallel to JDBC, but it is language-independent.

programs use JDBC to execute the SQL that queries and updates data stored in its MySql Database RDBMS.

5. Data Tier

The Data Tier is the layer of a Multi-Tier Architecture that stores the application data.

a. MySql Database

MySql[18] is a free[19] multi-user database server. It is ideal to use a database server in a Web-based project because several end users may use the site simultaneously. MySql supports the SQL query language used to query and update data stored in relational tables. MySql also supports TCP/IP connectivity. Unlike some other popular database servers, MySql does not support transactional processing or referential integrity in its tables. The section below describes two of OAKDA's key database tables, outlining their structure and functionality.

(1) Indexed OWL-DL Content Table - This table contains an indexed version of OWL-DL ontology documents loaded into the OAKDA system. The information contained in the OWL-DL flat file are read, parsed, vetted and processed by the Racer server and loaded into the MySql index table by Java programs using JDBC and SQL Data Manipulation Language (DML) statements. Table loading occurs when the system administrator or an end user of the site wishes to add OWL-DL content to the OAKDA data store.

Each term contained in the ontology is referenced in a MySql database table that has the following columns: NODE_TEXT, NODE_TYPE, NAMESPACE and OWL_FILE. The matrix shown in Table 9 describes the Ontology Index Table.

[18] http://www.mysql.com/

[19] General Public License (GNU)

Field Name	Data Type	Description
NODE_TEXT	String	The name of the OWL-DL term
NODE_TYPE	String	Describes the OWL-DL type of the named term (Class, Property, Individual, Role, etc)
NAMESPACE	String	The URI of the OWL-DL schema
OWL_FILE	String	The name of the OWL-DL flat file
DATE_INDEXED	Date	The date on which the file was loaded into OAKDA

Table 9. Metadata Descriptions for Indexed OWL-DL Content Table

Table 10 shows an example data that would be stored in the "Indexed OWL-DL Content Table."

NODE_TEXT	NODE_TYPE	NAMESPACE	OWL_FILE	DATE_INDEXED
Person	class	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
GrandMother	class	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
GrandParent	class	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
MaleSex	class	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
GrandFather	class	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
Parent	class	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
Gemma	individual	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
Peter	individual	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112
Matt	individual	http://www.owl-ontologies.com/generations.owl	generations.owl	20051112

Table 10. Example data for Indexed OWL-DL Content Table

The data in Table 10 is a transformation of the OWL-DL content. The reason this is done in advance is to enable quick lookups during the term searching portion of the program. This can be done by Racer but it would be prohibitively slow and resource intensive if Racer was used to read and perform reasoning of all OWL-DL file content each time search is requested. It is far less memory and CPU intensive to search an indexed list since no reasoning is necessary for term search.

(2) Selected Search Term Table – This table stores the items selected by the end user during terminology discovery which is later applied to the Web search. Since the applet is running in a separate HTTP session, the user selections and HTTP session ID must be stored for later retrieval by the application. The following matrix, in Table 11, shows the description of the Selected Search Term table.

Field Name	Data Type	Description
SESSION_ID	String	The name of the HTTP session that launched the TouchGraph applet.
SELECTED_NODE	String	The text of the OWL element node
DATE_SELECTED	Date	The date/time when the item was saved in the table

Table 11. Metadata Descriptions for Selected Search Term Table

b. OWL-DL Ontology File

Web Ontology Files contain the representational data used by OAKDA to find connected terminology information. In OAKDA, only OWL-DL files are processed by the RACER engine.

6. OAKDA Client and Server Components

Table 12 delineates, from a physical standpoint, where each component in OAKDA's Architecture resides. This is intended to provide a quick summary of the physical location of each component of the system. The "System Component" column of the table lists each part of the OAKDA system. The "Client" column is the computer used by the end user of OAKDA to access the system. The "OAKDA Host" column is the computer that is the server for the system. The "3rd Party Server" column contains those computer resources that are leveraged by OAKDA via the Web.

OAKDA Components			
System Component	**Client**	**OAKDA Host**	**3rd Party Server**
Apache Tomcat App Server		√	
Apache Web Server		√	
Google			√
Google Web Services		√	√
HTML / CSS / JavaScript	√		
Java™ Objects	√	√	
Java™ Applet	√		
Java™ Servlet		√	
OWL-DL Ontology File		√	
Racer Server		√	
TouchGraph	√		
Web Browser	√		
MySql Database		√	

Table 12. OAKDA Component Physical Location Matrix

D. OAKDA PROTOTYPE PROCESS FLOW

In the previous section, the components of OAKDA were introduced individually. This section explains how OAKDA components interact and details the sequence of the interaction. The diagram in Figure 78 provides a synopsis of all the communication pathways of OAKDA. The diagram is divided into three sections to indicate network boundaries: the Server Side, where the OAKDA system is hosted, the Client Side, where the client computer and the end user of OAKDA reside, and the Internet, where Google™ web services are located. All interactions traversing these boundaries are TCP/IP based network messages. There are three different modes of communication between the software and hardware system components: TCP/IP, direct program calls between Java™ programs, and System I/O[20] and disk I/O[21].

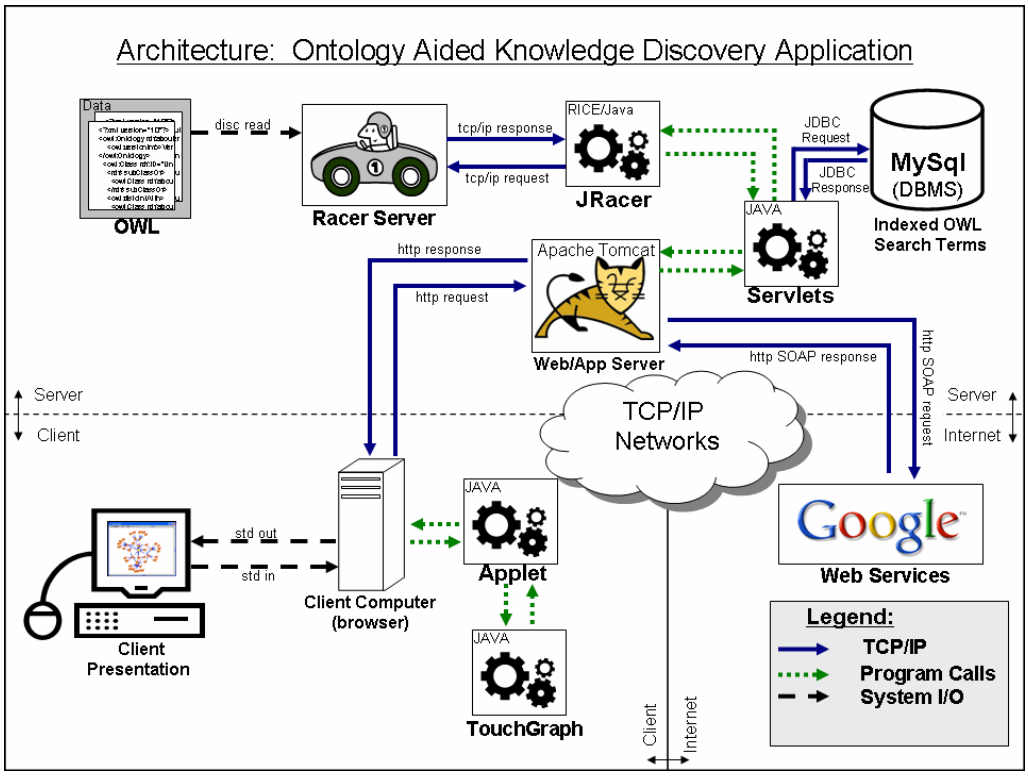

Figure 78. OAKDA Component Messaging Pathways

[20] These are standard input and output from the client computer, such as video, mouse clicks and keystrokes.

[21] For Racer's read of the OWL-DL files.

1. Anatomy of an OAKDA Search

In order to illustrate the complete workings of OAKDA, shown in Figure 1, this section describes a typical OAKDA usage scenario. This description touches upon all the components shown in Figure 78 and provides an overview of all interactions and processes underlying the system.

Firstly, it is assumed OAKDA will be used to refine Web search or to discover knowledge about a domain. We will suppose a user exists who has a search topic of interest in cartoons and an initial search term: "Jerry." The user launches the OAKDA home page to begin the process. Figure 79 shows that this HTTP requests/response communications is initiated with the Apache Tomcat Web server. The home page is composed of static HTML content. Figure 80 shows the OAKDA home page.

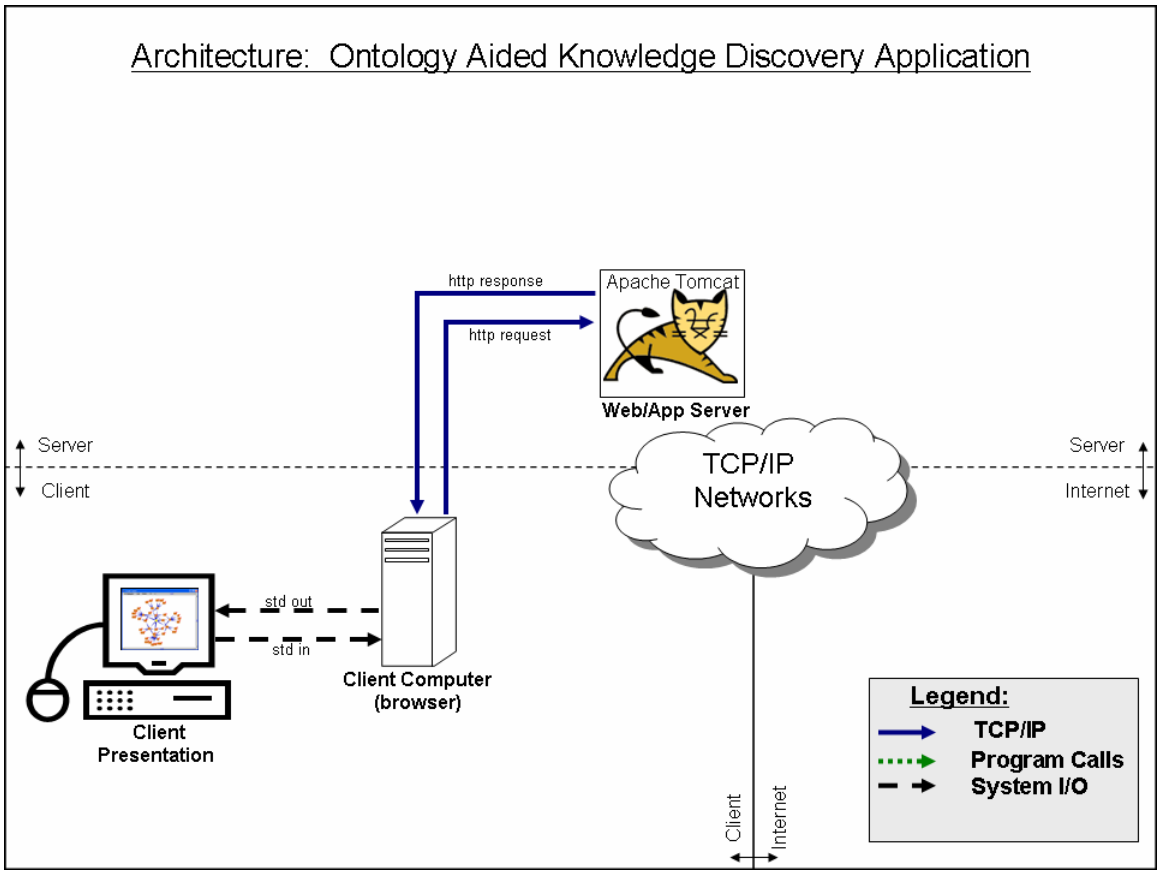

Figure 79. OAKDA Client / Web Server Interaction

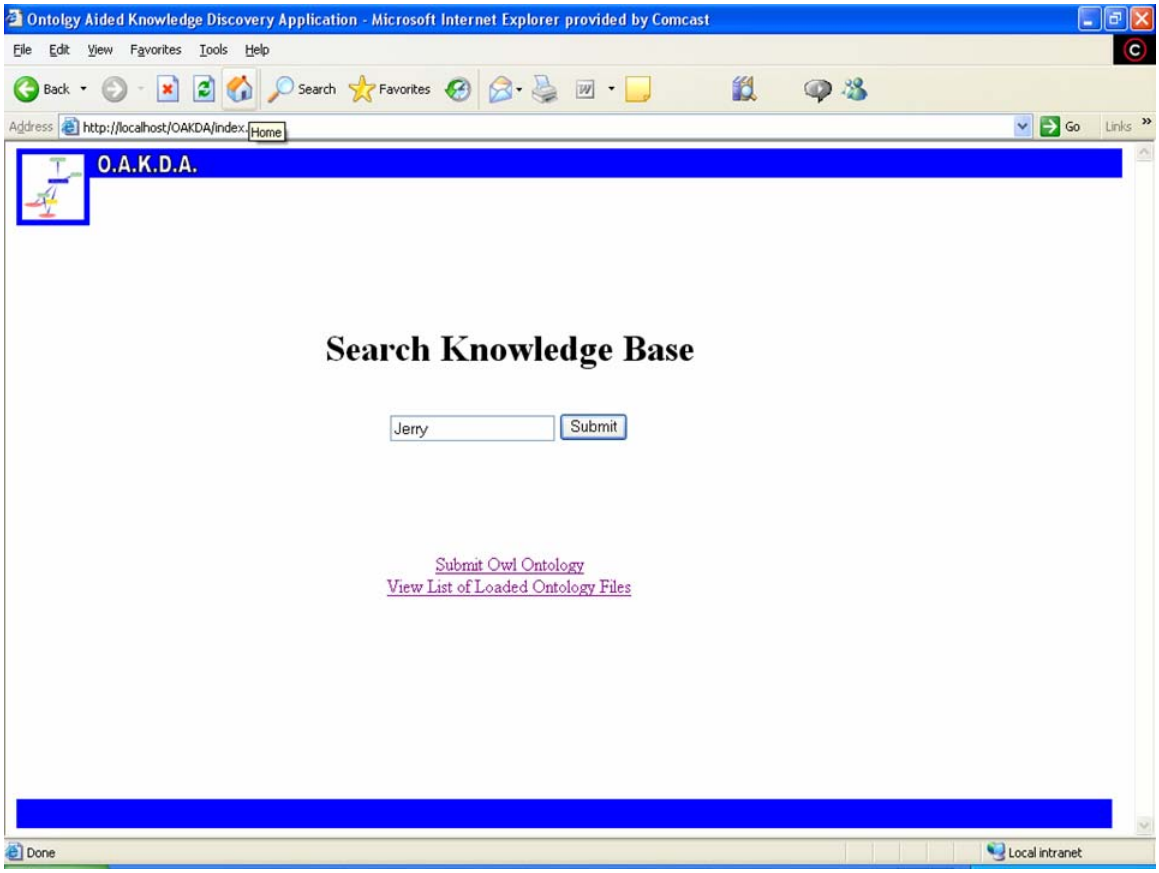

Figure 80. OAKDA Home Page

Next, the user enters the search term, “Jerry”, into the client side HTML form and presses the submit button. This action generates a HTTP request containing the search term as an argument to the Web server which is then dispatched to the Tomcat application server. Tomcat instantiates a program event that is handled by a specific Java servlet. The event invokes a Java program to create a SQL string, with search term “Jerry” incorporated, to query the MySql Database. Messages between Java and MySql are accomplished via the JDBC API. The database table queried contains records of indexed ontology content. The query returns the list of ontologies where the search term is found. The query results are passed back to the Java servlets where they are ranked by match closeness, incorporated into a HTML document and sent to the client via HTTP by the Web server. In this case, the term “Jerry” was found in the OAKDA database in an ontology called “Cartoon Star”. This interaction is depicted below in Figure 81.

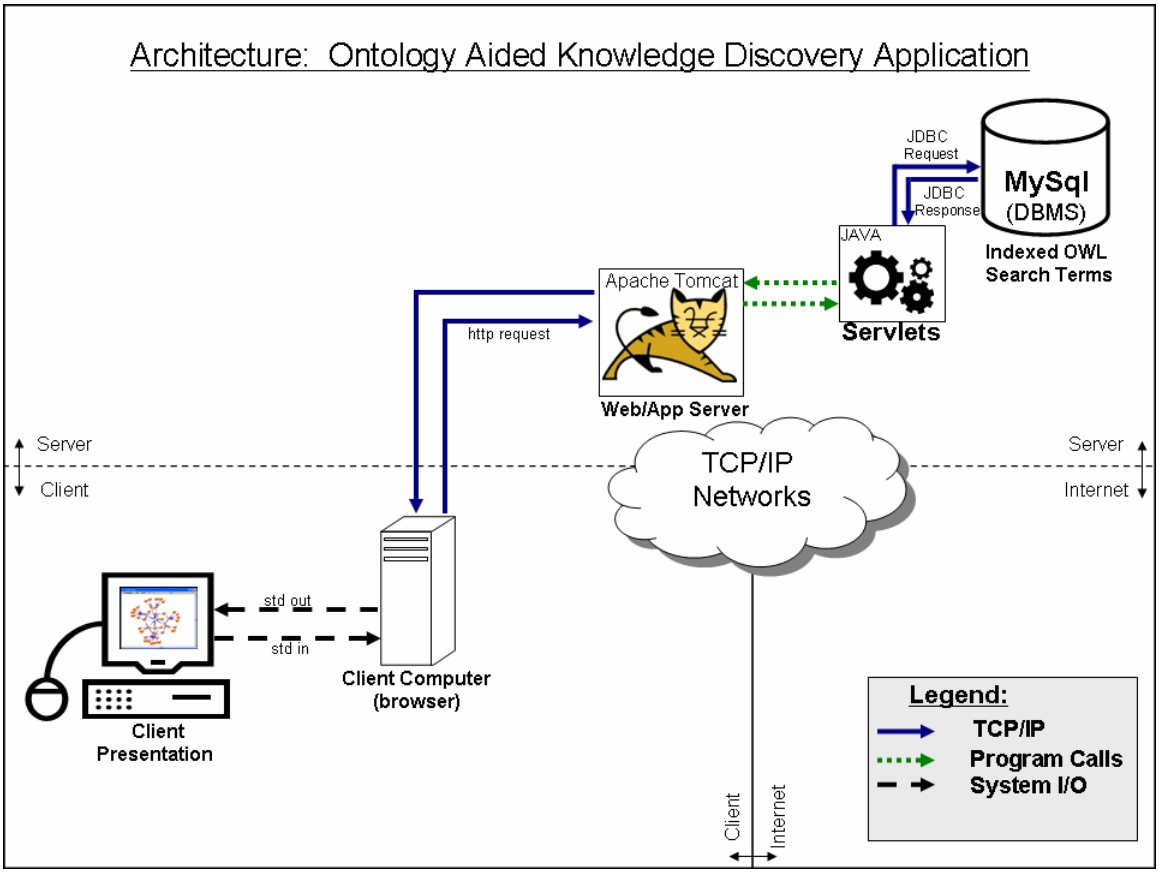

Figure 81. Client interaction with OAKDA Database

At this point the client presentation is showing a ranked list of found ontology content and a links to the source OWL-DL files where each match was found. The user selects the "Cartoon Star" ontology by pressing a button next to each result row. This action sends HTTP messages to Tomcat which are dispatched to Java servlets that create an HTML document to be sent as a response back to the client browser. HTML <APPLET> tags in the response direct the browser to download and instantiate a Java applet running on the browser's Java Virtual Machine[22] (JVM). The Applet is used to render OWL-DL data in the form of a directed graph for the user interface using TouchGraph software. The first action the applet does is to send an HTTP request message to the Web server which in turn accesses Racer via the JRacer middleware. Racer is instructed to read the "cartoon_star.owl" file and query it for all nodes directly associated to "Jerry." Description logics based reasoning executes as the ontology is

[22] A component of the Java platform which executes Java Bytecode.

instantiated in Racer's memory. [*See appendices 1, 2 for sample code*] A list describing all ontology information connected to "Jerry" is passed back to the Java Servlet which instantiates a Java bean object to be used as a container for this information. The Java bean is serialized[23] and sent, via the Web server, back to the client applet as HTTP data. The Applet then un-serializes the data back into a Java bean object. This object is passed to the TouchGraph program and used to render the nodes and edges of the directed graph depicting the OWL-DL ontology on the end user's client presentation. This chain of communications is shown in Figure 82.

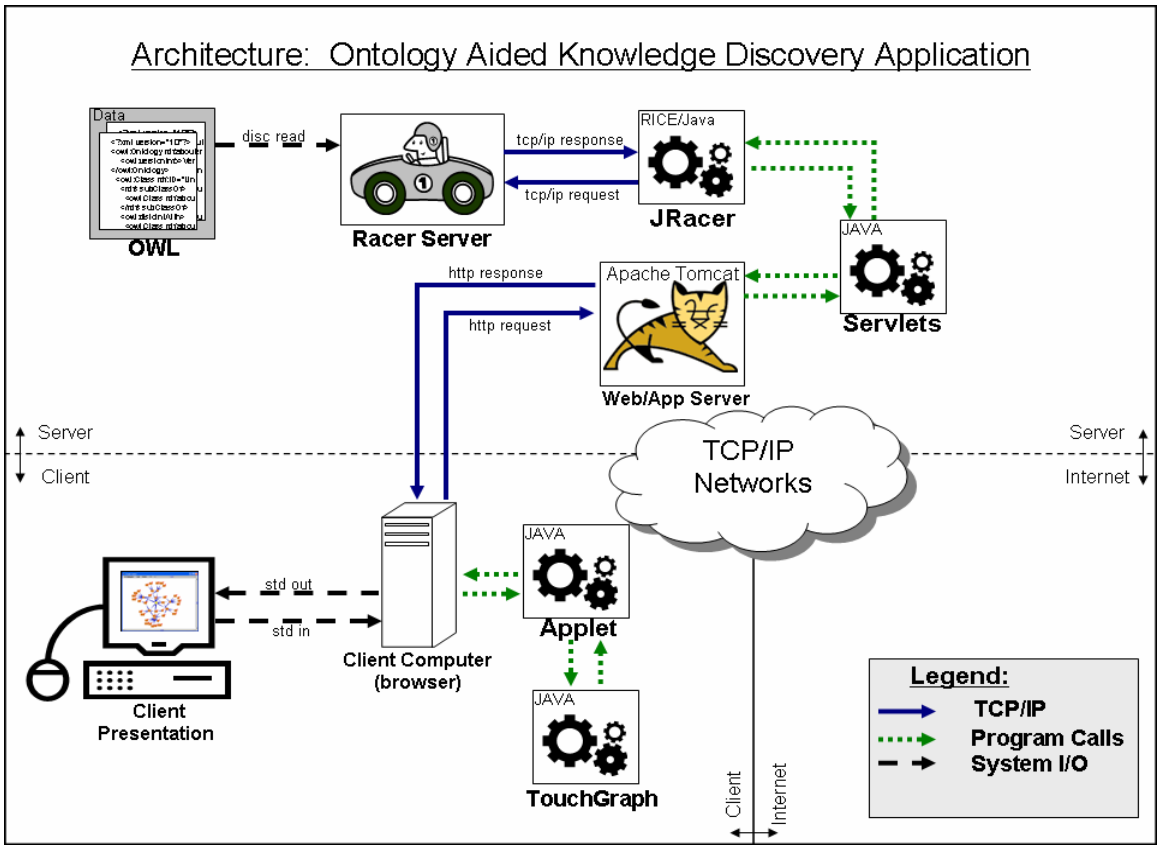

Figure 82. Client Applet Interaction with Racer Server

[23] Serialization is the process of saving an object onto a storage medium in order to later be able to re-create an object that is identical in its internal state to the original.

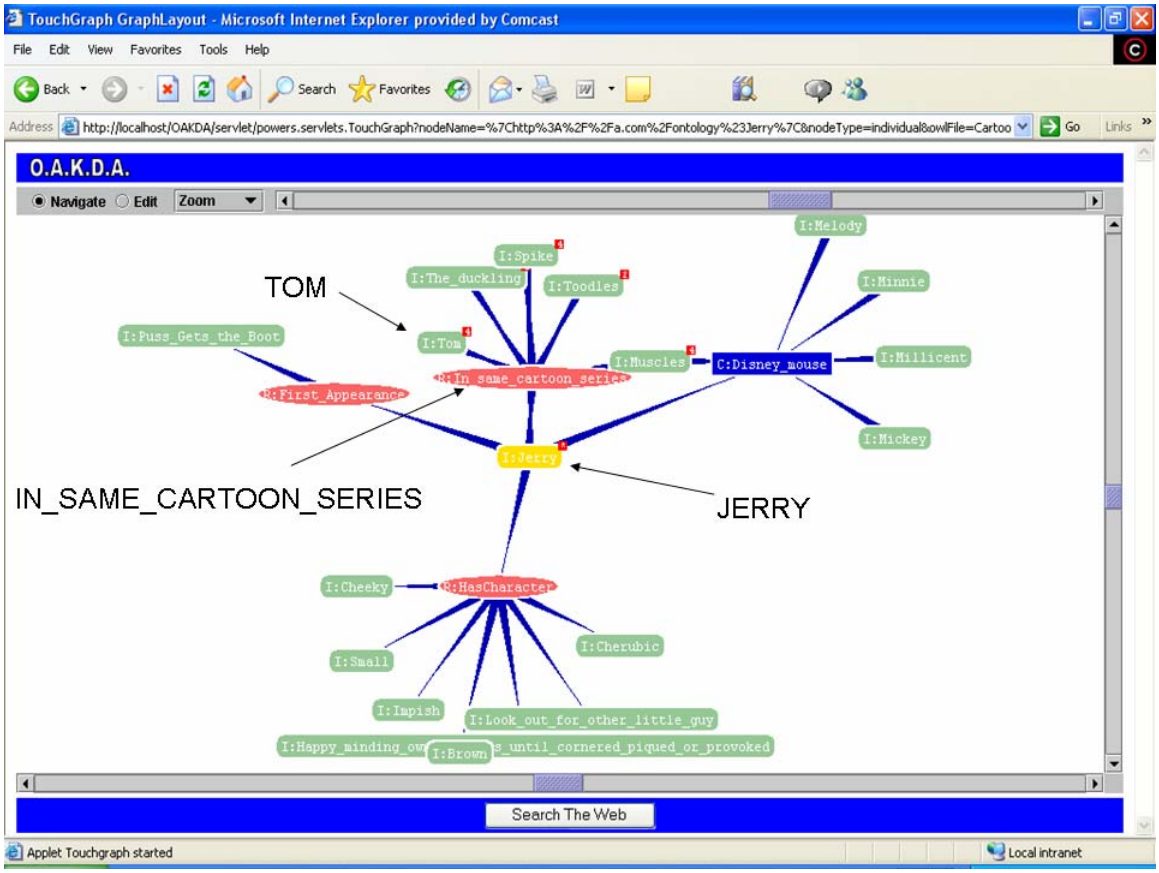

Figure 83. Graphical Visualization of the Cartoon Star Ontology

The client interface is now displaying a directed graph showing all the ontology nodes connected to “Jerry” in the Cartoon Star ontology (Figure 83). Some of the data in the display shows that “Jerry” is an Individual node instantiated from the “Disney_Mouse” class and that it is related to another individual named “Tom” through the “In_Same_Cartoon_Series” property. The user initiates an event to re-orient the graph around another node. By right clicking the individual, “Tom”, and selecting the “orient about node” option, the applet sends an HTTP message to the Web server that in turn dispatches an event to a Java servlet which, via JRacer middleware, queries Racer for all the OWL-DL data directly related to the “Tom” node. The data is incorporated into a Java bean, serialized and sent back to the client applet. The applet then re-displays a new directed graph centered on the individual, “Tom.” Some of the changes to the screen now show that “Tom” was instantiated from the “Disney_Cat” class, but the “Jerry” node is still present since it is related to “Tom” by the

"In_Same_Cartoon_Series" property. The components used for this event sequence are represented in Figure 83.

When the end user discovers the terms needed for the Web search in the displayed graph, the term is selected by right clicking on the node and choosing the "add as search term" option on the "Tom", "Jerry" and "Disney_Cat" nodes. This event causes the applet to send an HTTP message to the Web server dispatching a Java servlet to create a SQL "insert" statement containing the selected ontology values along with the session ID of the current HTTP session. The session ID is later used to retrieve the entire set of user selected search terms that are stored during that session. The servlet executes the SQL statement using JDBC API in the MySql database. The end result is a record added to a database table used to store user selected search terms. These component interactions are shown in Figure 84.

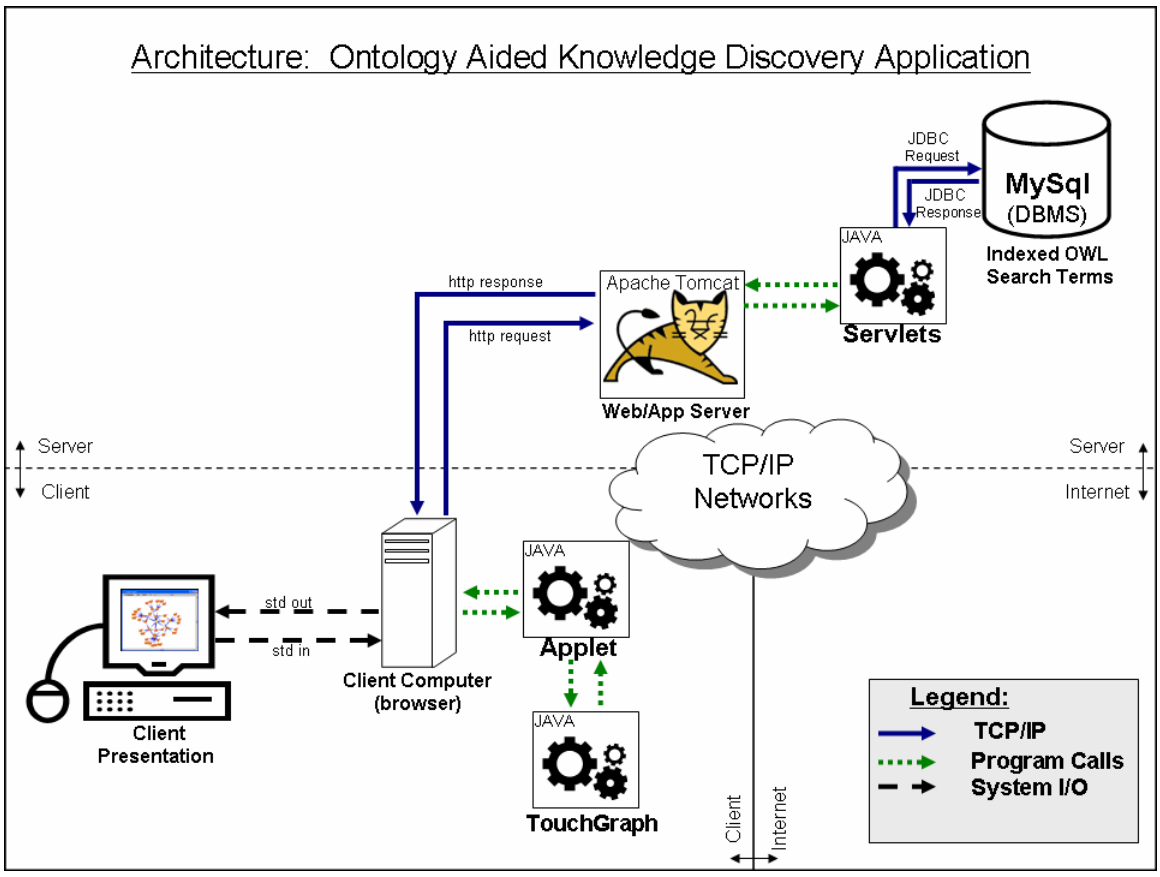

Figure 84. Interaction between Client Applet and Database

When the user has chosen search terms and pressed the button labeled “Search the Web” on the applet presentation view, an HTTP request is sent to the Web server which dispatches a servlet to create a SQL statement that accesses all the stored search terms chosen during the current Web server session. The servlet executes the query, retrieving “Tom”, “Jerry” and “Disney Mouse” from the database. This data is incorporated into an HTML document that is sent back to the client browser for display. The applet running in the browser’s JVM is terminated. The presentation now consists of the list of selected search terms in an HTML form designed to allow the end user to edit or modify the content. This action uses OAKDA components shown in Figure 81.

The user is now ready to initiate the Web search. By pressing the “Submit Query” button on the client presentation, HTTP messages are sent to the Web server which dispatches a Java servlet to incorporate the message data into a SOAP envelope destined for Google’s Web Services portal. The Web server routes the SOAP message request and receives Google’s SOAP HTTP response. Contained in the response is the listing of Web search hits that Google found in its database. This information is integrated into an HTML document and sent back to the client browser by HTTP. This interaction is shown in Figure 85 below. The OAKDA screen shot of the web links retrieved by Google Web services is depicted in Figure 86.

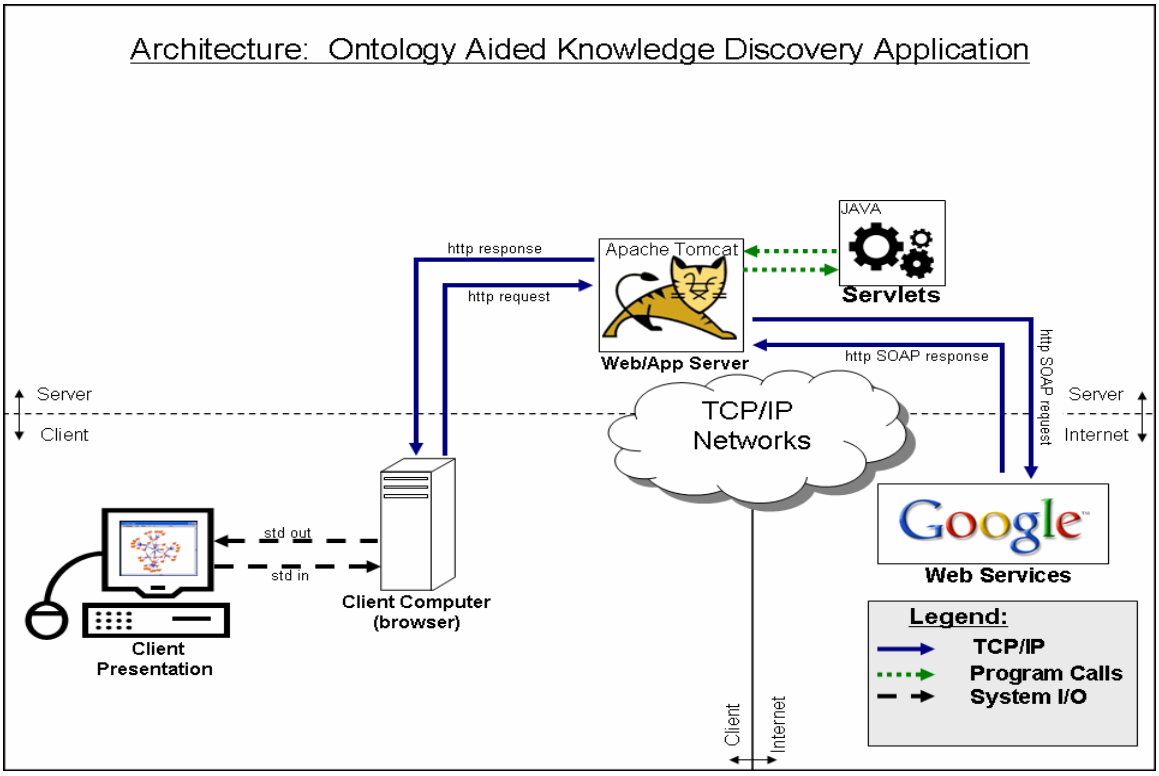

Figure 85. Client Interaction with Google Web Services

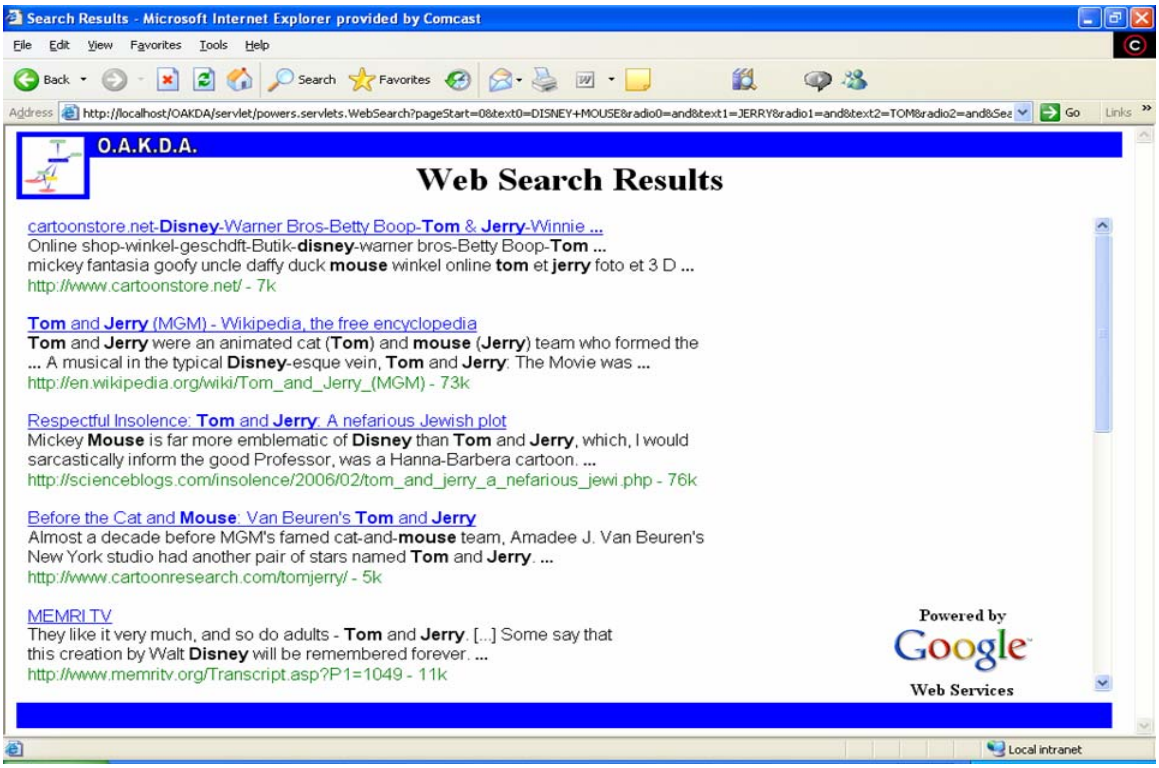

Figure 86. OAKDA Web Search Results

E. OAKDA PROTOTYPE EVENT SEQUENCE AND PROCESS FLOW

This section elaborates on the previous section and provides Sequence diagrams of the interactions of OAKDA processes when initiated by client generated events. A sequence diagram is a UML graphical construct used to show the sequence of interactions between object instances in a software based system. A sequence diagram is specific, referencing the method name of the function called in the interaction. The specific type of sequence diagram is a useful tool for a programmer for implementing a UML specification. For the purposes of this thesis, a more general type of sequence diagram will be used, called a Service Level Sequence diagram, which shows logic in detail but does not reference specific program function points. Following the diagrams and other depictions show a clear picture of the operation of the system.

1. OAKDA UML Sequences

a. Home Page

This section describes the events and programs that launch OAKDA's Home Page, the first step taken by the end user of the system. Figure 87 shows the associated sequence diagram.

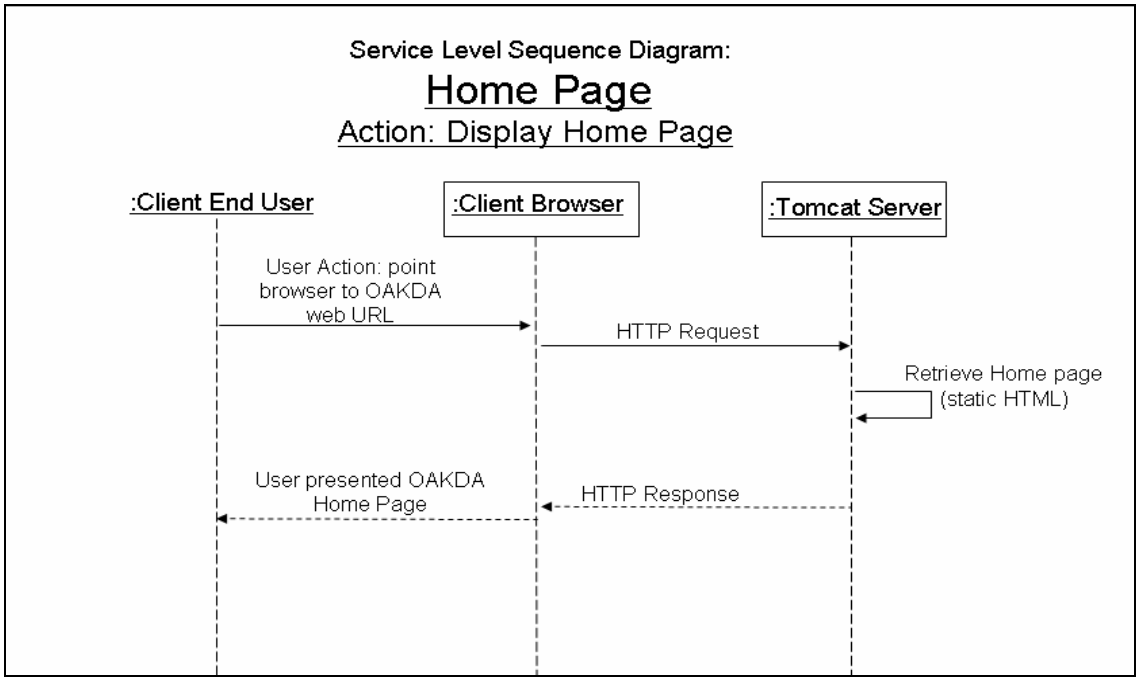

Figure 87. Home Page Sequence Diagram

In the starting state for this sequence, the client has not yet invoked OAKDA. The necessary preconditions are the end user has an Internet connection and Web browser. The sequence initiates when the end user launches a Web browser and enters the Web address for OAKDA. The browser sends a HTTP request to the OAKDA Web server. The Web server deciphers the request and sends the contents of the Index.html document back to the client as an HTTP response. The browser receives the response and renders the HTML to show the OAKDA home page on the client Web browser.

b. Ontology Search

This section describes the system events and programs that are activated when the end user initiates a search of the indexed OWL-DL content stored in the OAKDA database. The end result of the action yields a display of the search results in a ranked list. Figure 88 below shows the associated sequence diagram.

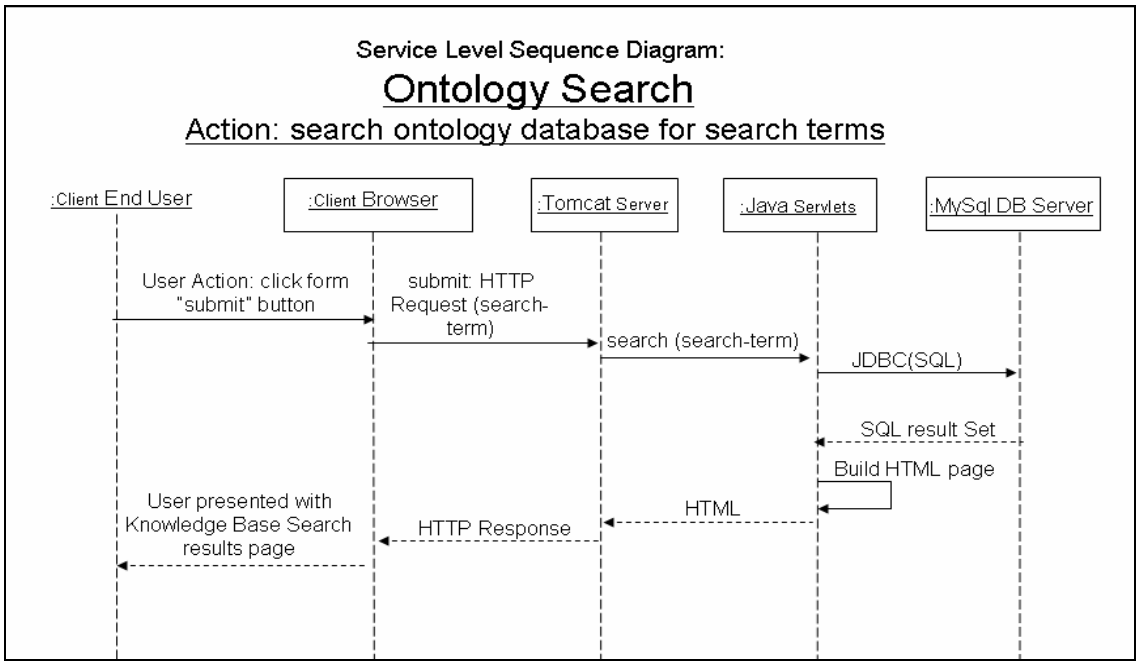

Figure 88. Ontology Search Sequence Diagram

In the starting state, the OAKDA home page is displayed. The home page consists of a form used to search the indexed OWL-DL content. There are no other

necessary preconditions. The process initiates when the end user enters search terms into the form and clicks the submit button.

Next, Tomcat Server receives a HTTP request and dispatches a call to a Java Program to compile and execute a SQL query against the data in the OWL_NODE_INDEX table in the MySql database. The query looks for partial or full string matches. For each record in the result set of the query, a score is computed that is later used to rank the closeness of the match with other rows. The data from the search is sorted by rank and formatted into the HTML content on the Search Results Page displayed on the client browser.

c. Ontology Search Results Page

This section describes the events and programs that construct the applet interface used for ontology navigation and terminology discovery when an OWL-DL file is selected. Figure 89 shows the associated sequence diagram.

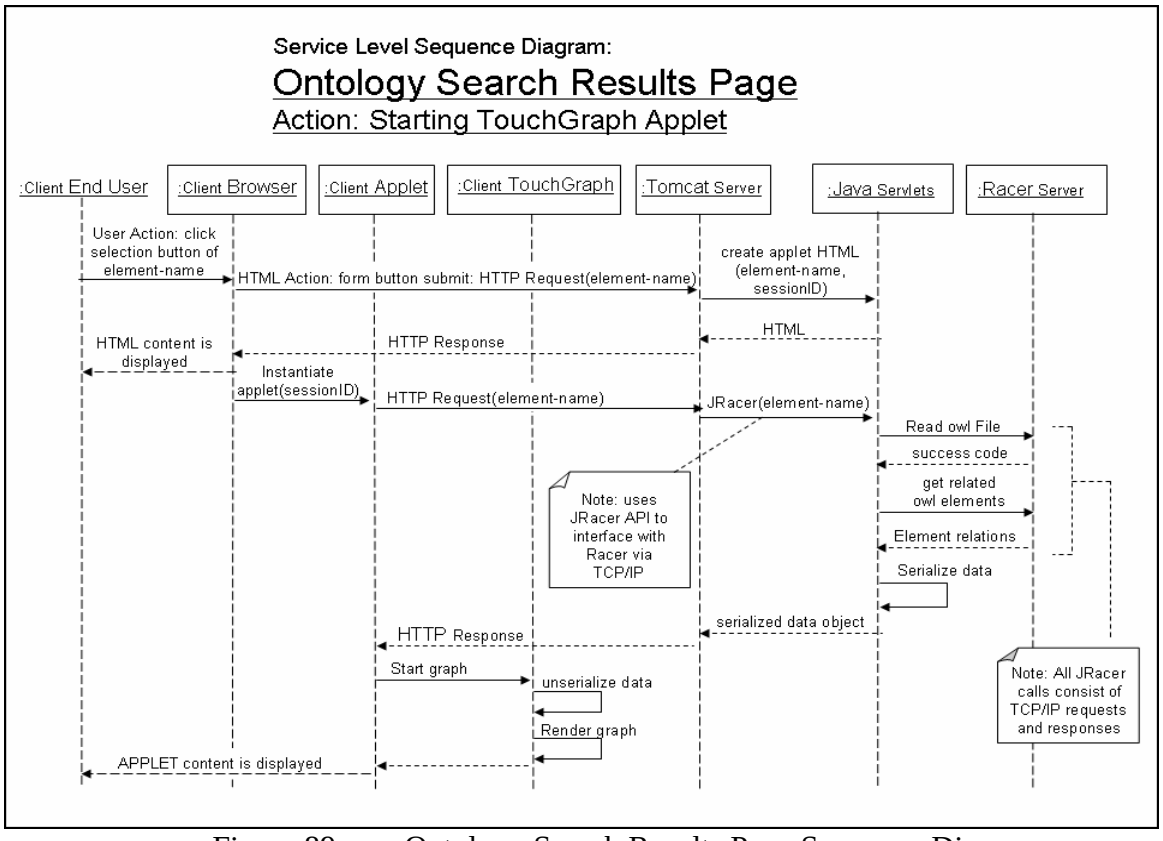

Figure 89. Ontology Search Results Page Sequence Diagram

In the starting state for this sequence, the Ontology Search Results Page is displayed. The page shows a sorted list of ontology nodes resulting from the Ontology Search Page. The list of the matched content from the search is ranked by the closeness of the match. The columns in the table are:

a. Score – A real number between 0 and 1 that describes the closeness of the match. This list is sorted in ascending order.
b. Element Name – The OWL-DL element name of the node in OWL-DL content.
c. Type – The element's OWL-DL type can consist of a CLASS, PROPERTY or INDIVIDUAL.
d. File – The name of the OWL-DL file where the element name was found.

The sequence starts when the user selects one of the buttons adjacent to a list item. At the end of the sequence the user is presented with graphical rendering of the ontology data in the form of a directed graph showing all nodes closely linked to the selected search term.

First, the element name, OWL-DL type, and OWL-DL file are passed as HTTP arguments to the Tomcat application server. Tomcat calls a servlet that builds an HTML response containing a <APPLET> tag reference to a Java™ program with the HTTP session ID and the OWL-DL arguments embedded in the page. The HTTP response is sent back to the client side browser. In rendering the HTML, the client browser is instructed to download the Java™ applet code from the Web server. The applet establishes a second HTTP session, sending the OWL-DL arguments to the application server.

Tomcat receives this message and responds by invoking a Java program that uses the JRacer API to communicate with the Racer server. The OWL-DL element names are used as arguments in the invocation. The first action performed is to check if the referenced OWL-DL file has already been loaded in the Racer server's memory. If it is not, a call is made requesting Racer to read the OWL-DL file. Racer's internal programs cause it to load, parse and execute description reasoning algorithms.

After the program returns a message that the Racer has successfully loaded the OWL-DL markup file, the servlet program performs a series of calls designed

to query Racer for elements directly related to the OWL-DL element name in the query statement. There are three types of queries that can be made. These are differentiated by their element type which can be one of the following: CLASS, PROPERTY, or INDIVIDUAL. This is shown in Figure 90, 91 and 92, respectively.

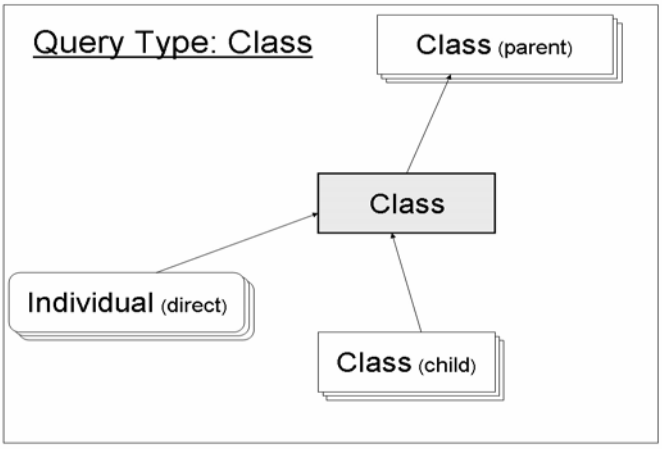

Figure 90. OWL Elements Directly Related to CLASS Type

When the OWL-DL argument of the query is a CLASS type, as in Figure 90, the servlet will query Racer for parent classes, sibling classes, child classes and direct instances. The query element "Class" is the central element of the graph. Arrows showing the inheritance relationship by their direction, link related elements. These queries only vary by the OWL-DL "type" of the argument.

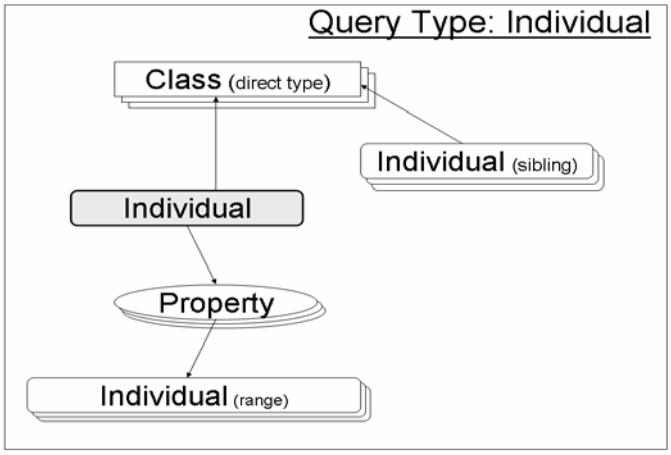

Figure 91. OWL Elements Directly Related to INDIVIDUAL Type

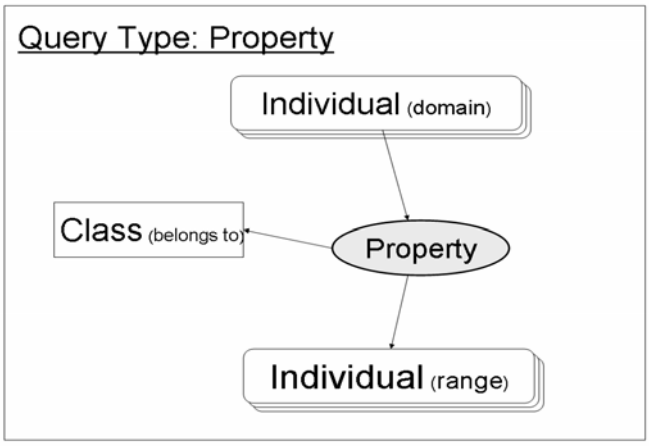

Figure 92. OWL Elements Directly Related to PROPERTY Type

Figure 91 and 92 depict the elements directly related to queries on the OWL-DL INDIVIDUAL and PROPERTY type, respectively. The query output is arranged into list that captures the pair wise relations of the element names. This list is contained in a Java™ bean[24] object and serialized[25] to "freeze" the object in its current state.

Tomcat routs the serialized output back to the client applet via HTTP. The client applet reconstitutes the data into a Java™ bean with the same state it had on the server side. The bean is passed as an argument into TouchGraph program methods that render a graphical representation of the relation pairs in the user interface. The element names become the labeled nodes in the two-dimensional graph and the inheritance relationship of the related nodes determines the arrow direction of the edge that connects them. The user interface displayed at this point is used for ontology exploration and selection of node names for a later Web search.

d. Ontology Exploration Applet GUI

This section describes the events and programs that enable the end user to traverse an ontology displayed by the TouchGraph Applet. When the user selects a new node as the focal point of the ontology, the graph reorients about that node. Figure 93 shows the associated sequence diagram.

[24] A Java Bean is simple Java Class that has "set" and "get" methods for each of its properties.

[25] "Flattening" a Java object into a persistent format such as a file or stream object.

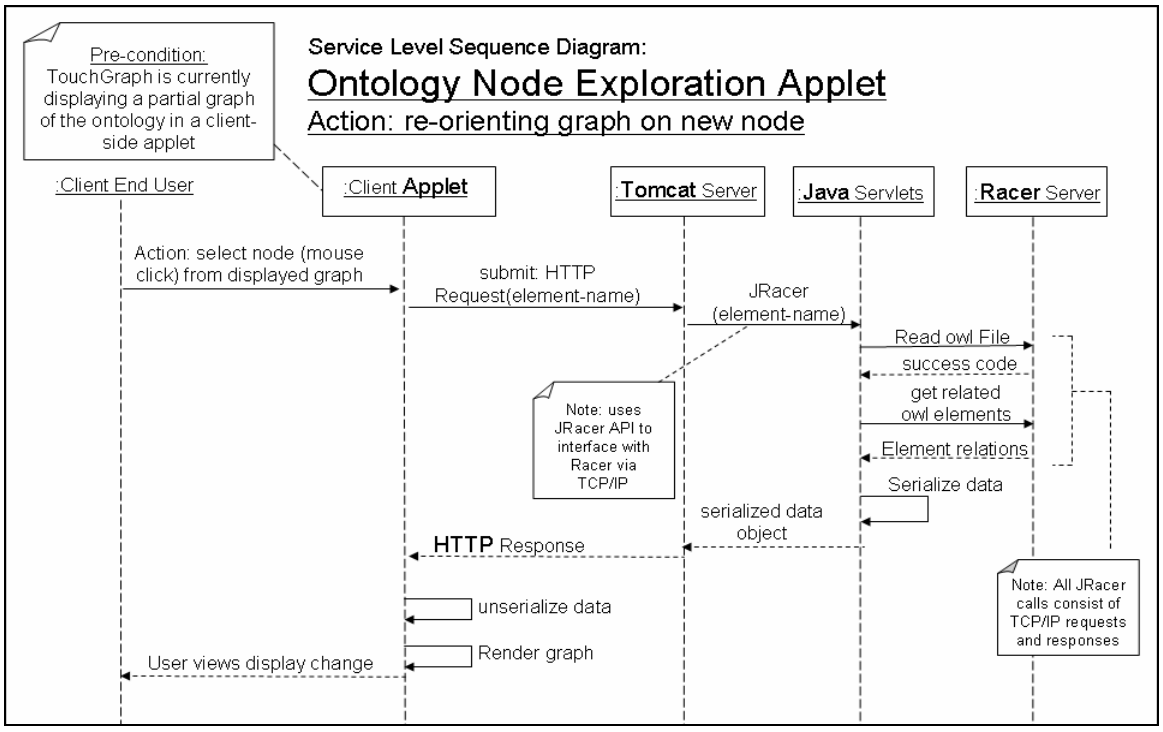

Figure 93. Ontology Exploration Function Sequence Diagram

The starting state is for the applet to display a portion of the ontology in the TouchGraph applet interface. When the user double-clicks on a given node, or right-clicks on a node and selects "orient about node", the applet will redisplay with the selected node as the focal point of the graph.

When the listener in the TouchGraph applet detects the mouse event, it responds by sending a HTTP message to Apache/Tomcat server. The message has HTTP parameters which specify the element name selected, the element type, and the OWL-DL file to which it belongs. Tomcat detects the messages and invokes Java™ programs using the JRacer API to communicate with the Racer server. The program creates TCP/IP calls to query the Racer server for all OWL-DL elements related to the new node. Just like in the section above, the information is passed down to the client applet and redisplayed on the user interface.

e. Ontology Node Selection

This section describes the events and programs that enable the end user to select a term from the displayed ontology to be used for Internet search. Figure 94 shows the associated sequence diagram.

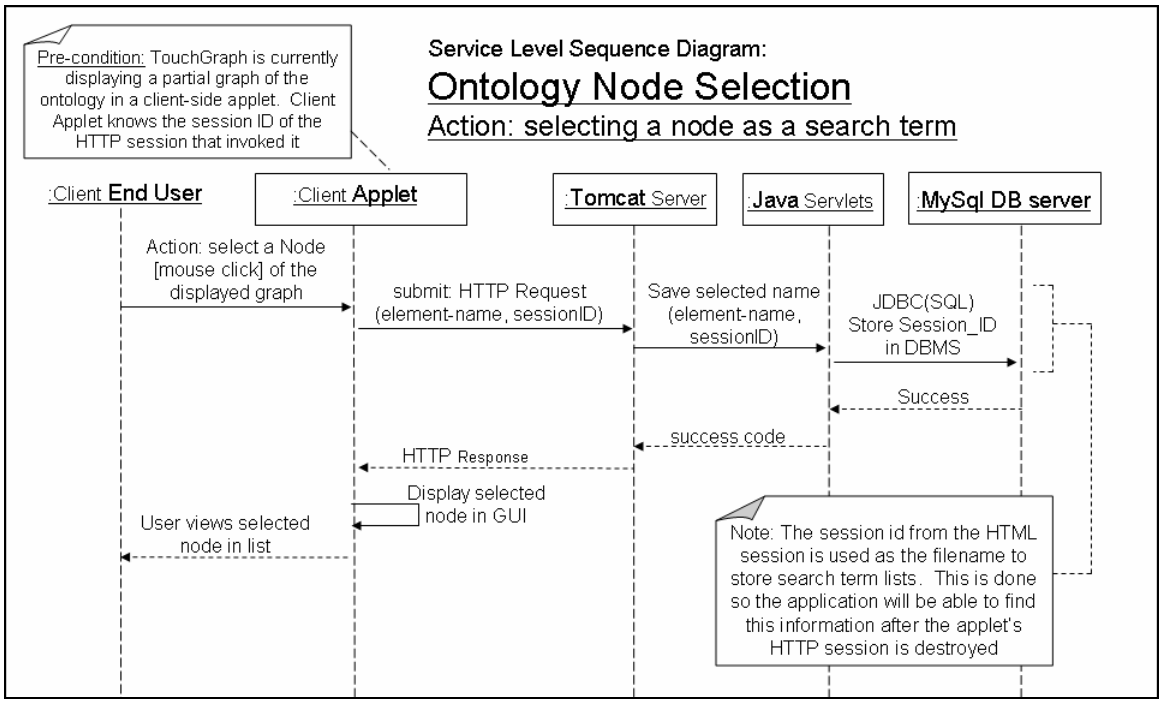

Figure 94. Ontology Node Selection for Search Sequence Diagram

In the starting state, the TouchGraph applet is displaying on the client GUI. There are no other preconditions for this sequence. The initiating action starts when a user right-clicks on a node and selects "add as search term." At the ending state, the selected term is stored in the OAKDA database with a reference to the HTTP session ID of the HTTP session that launched the applet and the TouchGraph applet displays the terms in a list box.

When a listener on the client side applet detects the mouse event, HTTP messages are sent to the Tomcat server. Tomcat then dispatches Java™ programs which invoke a SQL INSERT statement via JDBC to the MySql database server. The SQL statement directs the database to store the selected term and its HTTP session ID to the SELECTED_SEARCH_TERM database table.

f. Search Pre-Processing Page

This section describes the events and programs that construct the Web page which enables the end user to "fine tune" the Internet search query. Figure 95 shows the associated sequence diagram.

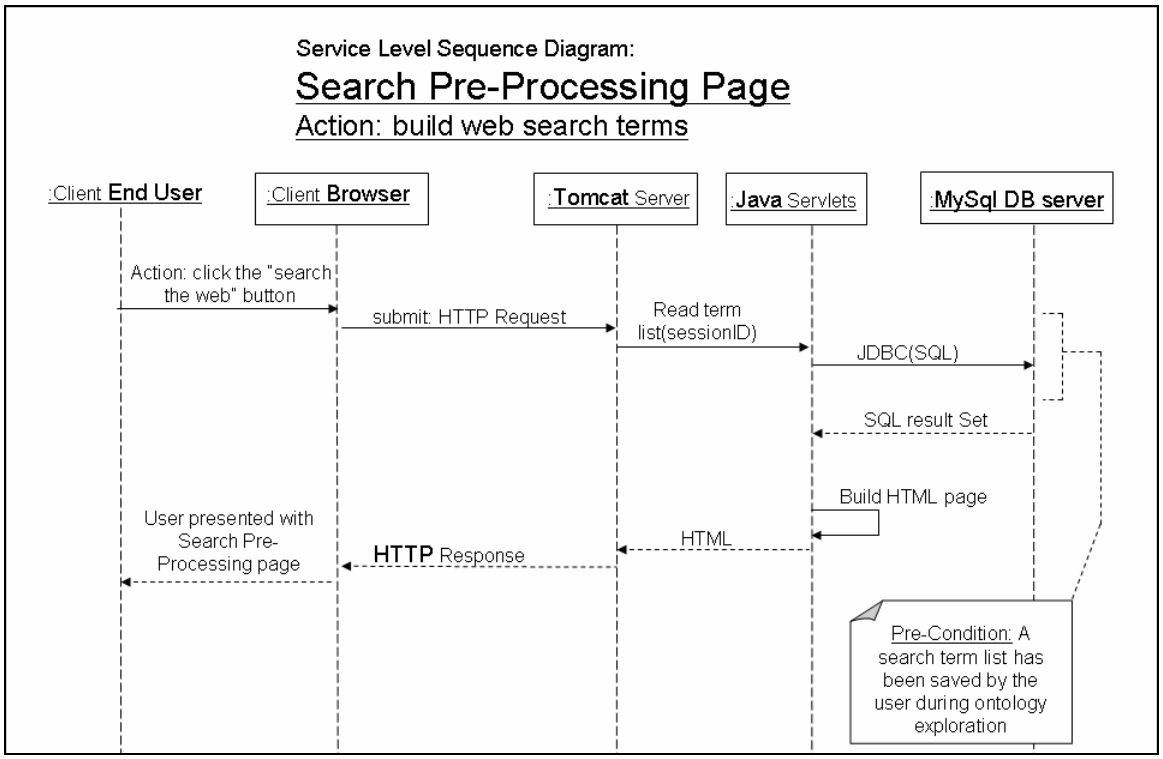

Figure 95. Web Search Pre-Processing Page Sequence Diagram

In the starting state, the OAKDA screen shows the Ontology Exploration GUI. There is a necessary precondition that search terms related to the current HTTP session are stored in the MySql database. The initiating action for this sequence occurs when the end user clicks on the “Search the Web” button on the Ontology Exploration page. The ending state shows the user a Web page displaying the search terms in editable text boxes with radio button options by each term that enable the user to further configure the syntax of the Internet search.

After the “Search the Web Button” is pressed, an HTTP request is sent to the Tomcat server which invokes a Java™ program to fetch records stored in the database SELECTED_SEARCH_TERMS table. The SQL statement uses the HTTP session ID as the key to find the records connected to the session of current OAKDA user. Some post processing takes place to reformat the list of terms for the Web search. Since the terms are often concatenated together either with ‘_’ characters or by upper/lower case transition, programs are invoked to parse the strings from their OWL-DL format into their component words and convert them to upper case text. The namespace information

on each OWL-DL element is discarded. Table 13 lists examples of typical transformations.

Typical OWL-DL format		Web Search Format
|http://abc.org/owl#Animated_cartoon|	→	ANIMATED CARTOON
|http://xyz.net/owl#HooverVacuumCleaner|	→	HOOVER VACUUM CLEANER

Table 13. OWL Element Transformations

This list of search terms is incorporated into the HTML page and sent back to the Tomcat server to be returned to the client browser as an HTTP response.

The displayed page is used to further configure the search terms for the Internet query. The form has button controls used for configuring the search with syntax used for the Google™ Web portal. The HTML form for this page has editable text boxes, radio buttons, and a submit button. The text box contains the search terms selected by the user during ontology exploration. These terms may be edited by the user to change the word or correct their spelling. The radio buttons are used to control the way the terms are used in the Web search. They enable the user to choose one and only one of several options. The "AND" radio button choice is the default selection and represents a Boolean "AND" for the search. The "OR" radio button implies that the search will apply logically "OR" the term with any other term that has the "OR" option selected. The "NOT" selection formats the Web search query to find Web pages that do not contain the term. The "REMOVE" option ignores the search term so that it will not be incorporated into Web search at all.

This search will look for pages that have the string apple in the content but not COMPUTER				
SEARCH TERM	**AND**	**OR**	**NOT**	**REMOVE**
APPLE	◉	○	○	○
COMPUTER	○	○	◉	○

Figure 96. Example Logical "AND" Search Syntax

This Search will look for pages that have either APPLE or ORANGE but do not contain COMPUTER				
SEARCH TERM	**AND**	**OR**	**NOT**	**REMOVE**
APPLE	○	◉	○	○
ORANGE	○	◉	○	○
COMPUTER	○	○	◉	○

Figure 97. Example Logical "OR" Search Syntax

This search will look for pages that contain APPLE or PEAR. The term: ARTILERY is ignored and is not present in the Web search				
SEARCH TERM	**AND**	**OR**	**NOT**	**REMOVE**
APPLE	○	◉	○	○
PEAR	○	◉	○	○
ARTILERY	○	○	○	◉

Figure 98. Example of Logical "OR" and REMOVE Search Syntax

Figures 96, 97 and 98 above show examples to of how the search configuration form assists the end user to configure discovered content into a Web search query. Figure 96 shows an example using a logical AND & NOT to retrieve web pages that contain "apple" but not "computer." Figure 97 is similar to 96, except the documents retrieved should have either "orange" or "apple" found in the content but not "computer." In Figure 98, the selection of the REMOVE option nullifies the inclusion of the content into the query. This means the pages retrieved will have "apple" or "pear" in the content but the "artillery" string will have no bearing on the search. The form configuration is used as input to be translated into search syntax used in the Google Web services interface.

g. Google Web Search Page

This section describes the events and programs that create the Web page showing the Internet search links from the Google Web portal. Figure 99 shows the associated sequence diagram.

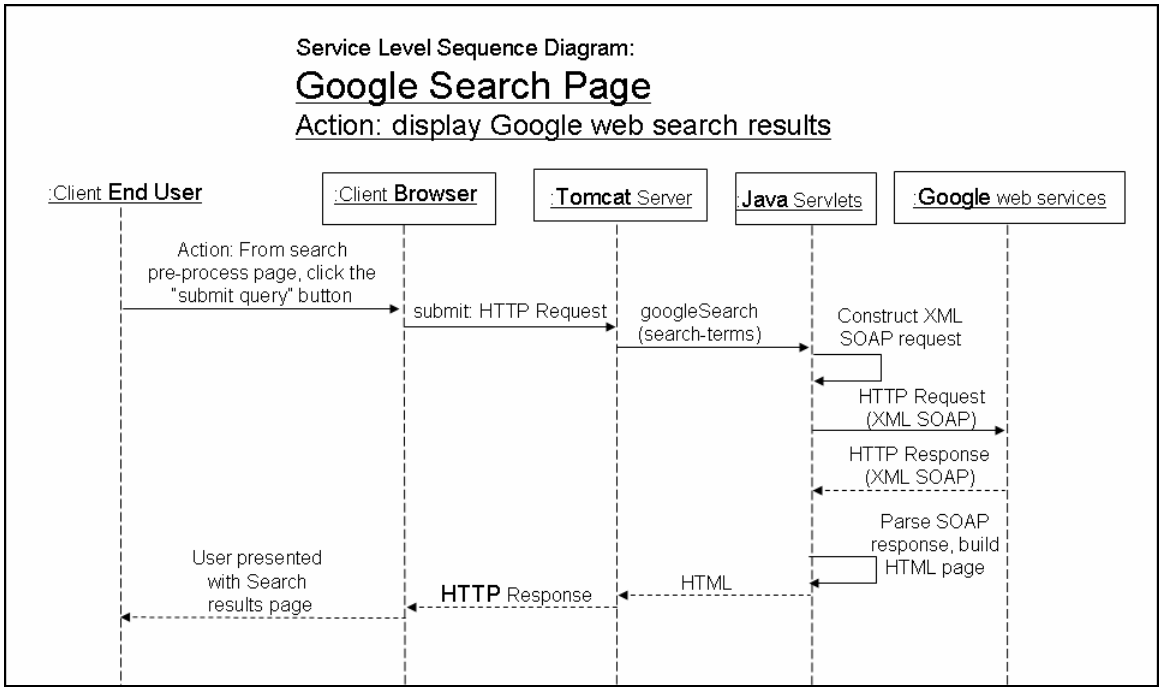

Figure 99. Google Search Page Sequence Diagram

In the starting state, OAKDA's client is displaying the Search Pre Processing Page. The action that initiates the process is when the user presses the "Search the Web" button on the Search Pre-Processing page. The ending state for this action displays an HTML page showing links to Web pages from Google™ Web Services that meet the criteria of the Internet search query.

After the initiating action, the client browser sends an HTTP request containing the Web search query in the Web form to the Apache/Tomcat server, which invokes the appropriate Java™ program for the next processing step. The program then constructs a Web services SOAP XML "envelope" to be sent the Google™ Web Services API using the HTTP protocol. Included in the envelope call are the formatted Web search parameter string and a registration key required for Google™ authentication. The

Web service sends back a SOAP response via HTTP that is dispatched by Apache/Tomcat server to programs which unwraps its contents. The data contained in the response is the very similar to content displayed in a typical Google™ Web site search. This data is incorporated into the HTML by the server side Java™ programs. An HTTP response containing the HTML document is sent back to the client Web browser. The view on the client is similar to a Google search result page. The user can access HTML anchors to visit the Web content found by the Google™ search.

F. CONCLUSION

The most significant aspects of the OAKDA are the Multi-Tier architecture, description logics reasoning services, and GUI visualization service that enables directed graph representation of the ontology. The bulk of effort and research for the development of OAKDA was spent in these areas. These three contributions to OAKDA are discussed in detail below.

1. Multi-Tier Architecture

The Multi-Tier architecture enabled an effective prototyping methodology to be employed in the development of each system component. As new methods and 3rd party software were evaluated, adopted or discarded, the Multi-Tier framework kept dependencies from effecting adjacent components. The architecture allowed extensive re-use of mature tiers through development iterations without the need for modification. This enabled tier integration and reintegration to be far less complicated then it could have been under a different architecture. Whenever there was a failure to separate tiers with a loosely coupled messaging framework, a penalty was paid in code re-writes when changes had to be accommodated.

The level of development effort to bring OAKDA to completion required almost 10,000 lines of Java™ code in 50 class files. Java™ should be mentioned as significant aspect of the architecture. Java™ was the cement used to integrate OAKDA's many components and was used to implement all business logic. Java's™ language support for networking (TCP/IP), useful utilities such as object serialization, and its ability to run on multiple platforms, enabled key design choices to succeed.

2 Racer

When development of OAKDA project first stated, the Jena API was evaluated as a possible middleware choice for the OWL-DL processing. The main reasons for interest in Jena were that the API was written in Java™ which was to be the language of choice for OAKDA's other components. After some effort to learn the API, it was found that Jena could parse and query OWL-DL effectively, but its reasoning services were difficult to discern. Jena documentation pointed to a capability for interfacing with 3rd party inference tools but the mechanisms for doing this were not readily understandable. At the time, Racer was being used as an add-in to the Protégé application to assist ontology development. Racer provided services to assist the developer to identify conflicts within an ontology and was used by Protégé to rearrange taxonomy structure to a more suitable form. It appeared that Racer could parse, query and reason with OWL-DL and the only obstacle to utilizing it in OAKDA was to find a method to incorporate it into the architecture. Racer's TCP/IP based interface allowed the software to work with any programming languages capable of forming and sending TCP/IP messages. Racer's native messaging framework uses a Lisp style syntax to query or publish to the Racer server. Fortunately, a small Java™ based interface for Racer was already developed, called JRacer, which abstracted TCP/IP communications and Racer's Lisp style syntax into Java function calls. Racer is server based software, which lends itself well to the Web server based architecture of OAKDA, since both Racer and the Web server need to handle simultaneous connections. After this discovery, Racer was selected as a component of the project design and the Jena API was abandoned. The incorporation of Racer and JRacer greatly accelerated the development of the system.

Racer had a certain drawbacks that may not be present with other inferencing middleware. For example, Racer has no straightforward method to query the property restrictions in class definitions. In OAKDA's visualization component, it may have been useful to show, along with the class name, the restriction statement that defines the attributes an individual must possess to be a member of the class. Instead, OAKDA only shows these restrictions when class instantiations, or individuals, are selected because the property restrictions are manifested as links between individuals. The effect of this on

the OAKDA system is that any OWL KB's without individuals do not reveal property restrictions and tend to appear as simple taxonomies. This seems to be a feature of Racer's description logic model. While restrictions can be published when a class is defined, it is difficult to query that information from the schema representation inside Racer's memory. The asserted content represented is easier to access. As a result, only in asserted content, are the class properties able to be expressed.

3. Visualization

It is difficult to visually decipher the patterns and meaning of large matrix of data without creating some type of representational image. Graphing techniques exist to help comprehend the meaning in a collection of data. OAKDA's usefulness to the end user greatly depended upon providing an effective visualization framework to amplify human cognition and navigation support of OWL-DL data.

OWL-DL's topology is best described as a directed graph to represent complex inheritance, and property relationships between the RDF resource nodes. In OAKDA's development, approaches using HTML/CSS did not yield good results because this format provided no easy means to mirror the topology of the OWL-DL documents. TouchGraph software is designed to represent directed graphs and thus readily able to depict OWL-DL. TouchGraph's capability to self-organize to fit the available screen space ensured the data was distributed evenly on the viewing area and did not obscure the content even when it was densely populated. TouchGraph necessitated the development of Applets running on the client browser which could communicate with the Web server.

The system was lacking in organizing the inheritance relationships in a way that could display directional relationships between nodes in an overarching pattern. Since the OWL representations of the KB contain cycles and bi-directional inheritance, the graph cannot be presented as a tree showing all inheritance relationships flowing from the top of the visualization area to the bottom. Sometimes this results in a complex jumble that has no clear top or bottom organization. However, this aspect of the visualization was present in most other techniques considered during the research. This may be due to the difficulty of visually displaying semantically rich ontology components and their relationships.

TouchGraph has an additional software development drawback in that it offers no API or service for importation of data into its system. TouchGraph's source code needed to be extensively modified to enable messaging with the OAKDA server, and GUI events had to be developed to enable navigation between linked parts of the OWL KB. The level of effort it took implement the modifications used to support OAKDA were not trivial.

Overall, the vision for OAKDA as a tool for a human end user to search, process and navigate OWL-DL ontologies was realized. It hoped that this application will be used as a tool for those interested in experimenting with the ontologies for knowledge discovery and Web search. For those researching OWL-DL processing technologies, the OAKDA application may be a useful as a baseline example.

VI. OAKDA VS. GOOGLE COMPARATIVE PERFORMANCE STUDY

A. INTRODUCTION

This chapter attempts to test and answer the main thesis research question: Can an ontology-based Web search application increase the effectiveness of Web search results over existing approaches for those searches that require a deep contextual knowledge of the domain of interest? This experiment compares the effectiveness of the results of Web search queries formulated by study participants using the OAKDA application with those obtained by the same participants using the widely popular Google search engine. The goal: to determine if OAKDA and the ontologies which it implements will have a significant positive effect on the precision and relevance of the web search queries generated by the participants in the experiment.

Effectiveness will primarily be measured along three dimensions: (1) whether the query retrieves pages containing the pertinent answer for the questions asked, i.e. the precision of the results; (2) whether the page retrieved is "about" the context that relates back to the ontology information domain, i.e. the precision of the context; and (3) the participant's own subjective rating on how well OAKDA and Google performed in answering the search tasks.

B. EXPERIMENTAL DESIGN

The experimental design used for the study is described in the following subsections, namely participants, apparatus, and data collection procedures.

1. Participants

The participants in this study consisted of 10 adults ranging from 25 to 70 years of age. About half of the subjects are information technology workers or have some involvement in IT an related profession. The rest are employed in other various fields. All have, to varying extents, a college education and experience in researching topics on the Internet.

2. Apparatus

All participants in the study used Microsoft Internet Explorer and a high speed Internet connection. The OAKDA application was used as the search environment for the test group (A full description of OAKDA and how it functions can be found in Chapter 6 of this thesis). The Google Web search portal was used as the only search environment for the control group.

3. Data Collection Procedures

The data points captured from each study participant were in two main categories: personal data and experiment data. The personal data included the participant's name, age, and profession. For this study, personal data was not considered in the data analysis.

Before conducting the experiment, each subject was trained in the use of OAKDA with a sample search task. The training covered OAKDA's user interface functions as well as recommendations on how to construct search queries. No training was given for using Google, but many of the same tips given for OAKDA were applicable for producing effective search results in Google. The participants were assumed to have extensive experience using Google, or Web search tools like it.

Each subject answered half of the Web query tasks using OAKDA (Test group) and half using Google (Control group). This method ensured that all participants took part in both the control and test groups an equal number of times. Each Web query was more or less equally assigned to the control and test groups[26].

For the control group, the study participants were allowed as many as three attempts to refine their search query. They were permitted a maximum of 5 minutes to work on each question. During this time, they were allowed to visit the retrieved web pages to find information that could enhance the subsequent queries. The query term list from each attempt was captured as a data point.

The test group was also allowed 5 minutes per questions but they were granted more time if they were experiencing technical problems or needed help with application

[26] For the rest of the chapter, the sets of control and test data (which are different depending on the participant) will be referred to as control and test groups.

functionality. No other type of help was given to the test subjects. OAKDA stored all submitted queries in a database and only the first three queries for each participant were used for the study data.

The experimental data captured for each participant were, for both control and test groups, the search query identifier, the number of the attempt, the Web query text, the participant's overall rating of the application performance for all queries, and the precision scores as calculated by the experimenter. One record was generated for every Web search query. Table 7.1 shows the data definitions for a recorded observation.

Field Name	Data Type	Description
Subject_Id	Integer	Unique numeric identifier for person participating in the study
Group_Type	String	Identifies data point as being in "TEST" or "CONTROL" study group
Question_Id	Integer	Unique numeric identifier for questions posted to the participant
Attempt_#	Integer	Search query attempt number (1..3)
Search_Query_String	String	Actual Google search term list
Answer Score	Number	Score between (0..1) equal to the number of the top 10 web hits generated by the Search_Query_String that contained the answer to the search task specified by the Question_id
Context Score	Number	Score between (0..1) equal to the number of the top 10 web hits generated by the Search_Query_String that contained the correct context for the search task's domain

Table 14. Definition of Experiment Data

There were six Web search tasks in the study. The participants were instructed to provide the information asked for by the task in the form of a web query that retrieves information as relevant as possible to the task's target answer. The search tasks assigned to the participants and the associated ontologies are listed in the table 7.2 below.

Question Id	Task text	Ontology File
1	Formulate a search query that will retrieve information about the geographic features of a tropical Island Nation off the coast of the African continent.	Geography.OWL
2	Formulate a Web Search query that retrieves the name of the actor(s) who voiced the dog character in the cartoon “Dog Trouble”	Cartoon_star.OWL
3	Formulate a search query which will retrieve web pages pertaining to at least two French artists who made both paintings and sculptures in the style of 19th Century Realism.	ArtHistory.owl
4	Formulate a search query which will retrieve web pages pertaining to a computer programming language that that can be written in either a "Procedural" or "Object Oriented" programming paradigm. This language is a derivative of the language used to develop the UNIX operating system.	Programming Languges.OWL
5	Formulate a search query which will retrieve web pages pertaining to an electric guitar that is the signature model of a famous musician. The musician’s first name begins with “L” and his career spans from the 1930’s to the present day.	Guitar.OWL
6	Formulate a query that retrieves web pages containing the name of a type of golf ball commonly used ~150 years before the introduction of the modern golf ball and the components materials from which it was manufactured.	Golf.owl

Table 15. Participant Search Tasks

After the participants finished creating Web queries for all the search tasks, they were asked to rate subjectively the effectiveness of both Google’s and OAKDA’s searches on a scale between 0 and 10. This data was intended to represent the participant’s perceived measure of OAKDA’s usefulness compared to Google’s, and represents the “Participant Rating” score of the experiment data.

Recall performance was not considered for the study since the data analyzed is “pulled” from a Web search portal, the total set of relevant data on the Web was too large to be efficiently measured. The precision scores were calculated by checking the relevance of the top *n* number of retrieved Web pages from any given query. In general, the precision was defined as the ratio of the number of relevant hits retrieved to the total number of examined hits. The statistic was expressed as a percentage. The queries recorded for each participant were given a rating score in two different analysis categories.

a. The "Answer Precision" Score

Each query result was rated between 0 and 1 (0..1) for the presence of the "answer(s)" to the search task in the text of the retrieved web hits. The examination of the Web pages was accomplished with a Java™ program which downloaded the pages and used regular expression processing to find the answer text within the page content. The top 20 Web hits generated by each query were examined. For each query, the answer precision score was determined by calculating the number of Web sites with the answer text divided by the total number of examined sites. A score of .75 meant the answer text was present in 15 out of the top 20 Web pages retrieved by the query.

b. The "Context Precision" Score

Similarly, each of the top ten retrieved Web pages was examined to assess if they had the correct contextual background pertaining to the search task. For example, if the search task pertains to the domain of golf equipment, and the retrieved web pages are about Gutta Percha tree agriculture, then the page was considered to be contextually incorrect. The determination of whether a given page was contextually accurate was achieved by a visual analysis. The context precision score was calculated as the ratio of the number of pages that had the correct context for the search task.

The scores for each query formed the raw data used for the statistical analysis. As stated earlier, each participant answered half (three) of the search tasks with Google and the other half with OAKDA. The ten participants in the study produced a total of 125 Web search queries in answering six Web search tasks (each was allowed up to three attempts). Therefore, each participant produced, on average, 2.08 queries out of a possible 3 for each test and control data set. Only the best performing query was kept for each search task answered, producing 30 total observations for the test group and 30 for the control group. Each participant's control and test group scores were averaged, leaving a total of ten observations for each experiment group, or 1 observation per participant.

This data summary was performed for both the "Answer" and "Context" data sets, leaving three total data sets for analysis: Participant Rating score, "Answer"

precision score and the "Context" precision score. The next section will describe the type of statistical analysis used to measure the data and the findings.

C. DATA ANALYSIS PROCEDURES

The methodology followed in the experiment yielded three data sets to be used as inputs for the statistical analysis. Table 7.3 below depicts the performance scores for the "Participant Rating," "Answer Precision," and "Context Precision" data sets.

	Participant Rating		Answer Precision		Context Precision	
Participant ID	**Google**	**OAKDA**	**Google**	**OAKDA**	**Google**	**OAKDA**
1	0.6	0.7	0.417	0.850	0.600	0.900
2	0.7	0.9	0.833	0.567	0.700	0.800
3	0.7	0.75	0.650	0.900	0.667	0.567
4	0.4	0.6	0.367	0.550	0.400	0.667
5	0.6	0.9	0.583	0.817	0.733	0.733
6	0.65	0.9	0.550	0.833	0.800	0.767
7	0.4	0.8	0.400	0.867	0.633	0.600
8	0.3	0.8	0.383	0.600	0.667	0.833
9	0.7	0.7	0.500	0.833	0.633	0.533
10	0.4	0.8	0.300	0.783	0.433	0.167
Mean score	0.545	0.785	0.498	0.760	0.627	0.657

Table 16. Experiment Data Results

The objective for the analysis of the data sets was to see if there is a significant difference between score means of the test and control data sets. The method used to test for significance was the *t* test dependant means. The *t* test involves a comparison of means from two different groups and focuses on the differences between the scores using the following formula:

$$t = \frac{\sum D}{\sqrt{\frac{n\sum D^2 - (\sum D)^2}{n-1}}}$$

where,

Σ*D* is the sum of all differences between groups

Σ*D* is the sum of all differences squared between groups

n is the number of pairs of observations.

1. *t* Tests for the "Participant Rating"

This section shows the eight steps used to compute the *t* test statistic. To reiterate, the "Participant Rating" scores consist of the participants' rating scores of the perceived effectiveness of OAKDA and Google in retrieving relevant Web pages to answer the search tasks

a. Statement of the Null and Research Hypothesis

The null hypothesis states that there is no difference in the participant rating between effectiveness score means of Google and the OAKDA searches. The research hypothesis is that the participants rate the OAKDA searches as more effective than Google searches. The research hypothesis is a one-tailed, directional research hypothesis because it posits that the OAKDA score will be higher than the Google score.

The null hypothesis is: $H_0 : \mu_{Google} = \mu_{OAKDA}$.

The research hypothesis is: $H_1 : \overline{X}_{Google} > \overline{X}_{OAKDA}$.

b. Set the Level of Significance or Type I Error Associated with the Null Hypothesis

The risk of Type I error or level of significance is 0.05.

c. Select the Appropriate Test Statistic

The appropriate test statistic is a *t* test for dependent means, also known as the *t* test for paired samples, since we are dealing with a group of scores for the same participants.

d. Compute the Obtained Value

The obtained value for t is:

$$t = \frac{\sum D}{\sqrt{\frac{n\sum D^2 - (\sum D)^2}{n-1}}} = 4.65$$

e. Determine the Value Needed for the Rejection of the Null Hypothesis

The degrees of freedom are n –1 or 9. Using this value and appropriate t-value tables [Salkind, 2004, 358], the value needed for rejection of the null hypothesis at the 0.05 significance level is 1.833.

f. Compare the Obtained Value to the Critical Value

The obtained t value is 4.65, larger than the critical value of 1.833 needed for rejection of the null hypothesis.

g. Decision Time

Since the obtained value is greater than the critical value, the null hypothesis is rejected. This indicates that users rate the effectiveness of OAKDA searches as, indeed, higher than that of a Google search for the experiment query set.

2. *t* Test for the "Answer Precision"

This section shows the eight steps used to compute the *t* test statistic. To reiterate, the "Answer Precision" data set consists of the precision scores of retrieved web pages, measured by the number of sites containing the answer text of the search task.

a. Statement of the Null and Research Hypothesis

The null hypothesis states that there is no difference between the answer precision scores of Google as compared with OAKDA. The research hypothesis is that OAKDA search results contain the "answer" to the research task more frequently than Google searches. The research hypothesis is a one-tailed, directional research hypothesis because it posits that the OAKDA precision will be higher than Google.

The null hypothesis is: $H_0 : \mu_{Google} = \mu_{OAKDA}$.

The research hypothesis is: $H_1 : \overline{X}_{Google} > \overline{X}_{OAKDA}$.

b. Set the Level of Significance or Type I Error Associated with the Null Hypothesis

The risk of Type I error or level of significance is 0.05.

c. Select the Appropriate Test Statistic

The appropriate test statistic is a *t* test for dependent means, also known as the *t* test for paired samples, since we are dealing with a group of scores for the same participants.

d. Compute the Obtained Value

The obtained value for t is:

$$t = \frac{\sum D}{\sqrt{\frac{n\sum D^2 - (\sum D)^2}{n-1}}} = 3.86$$

e. Determine the Value Needed for the Rejection of the Null Hypothesis

The degrees of freedom are n –1 or 9. Using this value and appropriate t-value tables, the value needed for rejection of the null hypothesis at the 0.05 significance level is 1.833.

f. Compare the Obtained Value to the Critical Value

The obtained t-value is 3.86, larger than the critical value of 1.833 needed for rejection of the null hypothesis.

g. Decision Time

Since the obtained value is greater than the critical value, the null hypothesis is rejected; indicating that according to the measure of retrieved Web pages containing the search task answer, the precision of OAKDA is higher than Google results for the experiment query set.

3. *t* Test for the "Context Precision"

This section shows the eight steps used to compute the *t* test statistic. To reiterate, the "Context Precision" data set consists of the precision scores of retrieved web pages, measured by sites containing the correct context of Web page content pertinent to the search task.

a. Statement of the Null and Research Hypothesis

The null hypothesis states that there is no difference between the "Context" precision scores derived from Google as compared with OAKDA. The research hypothesis is that OAKDA search results contain the correct research task context more often than Google searches. The research hypothesis is a one-tailed, directional research hypothesis because it posits that the OAKDA score will be higher than the Google score.

The null hypothesis is: $H_0 : \mu_{Google} = \mu_{OAKDA}$.

The research hypothesis is: $H_1 : \overline{X}_{Google} > \overline{X}_{OAKDA}$.

b. Set the Level of Significance or Type I Error Associated with the Null Hypothesis

The risk of Type I error or level of significance is 0.05.

c. Select the Appropriate Test Statistic

The appropriate test statistic is a *t* test for dependent means, also known as the *t* test for paired samples, since we are dealing with a group of scores for the same participants.

d. Compute the Obtained Value

The obtained value for t is:

$$t = \frac{\sum D}{\sqrt{\frac{n\sum D^2 - (\sum D)^2}{n-1}}} = 0.54$$

e. Determine the Value Needed for the Rejection of the Null Hypothesis

The degrees of freedom are n –1 or 9. Using this value and appropriate t-value tables, the value needed for rejection of the null hypothesis at the 0.05 significance level is 1.833.

f. Compare the Obtained Value to the Critical Value

The obtained value is 0.54, smaller than the critical value of 1.833 needed for rejection of the null hypothesis.

g. Decision Time

Since the obtained value is less than the critical value, the null hypothesis is upheld, indicating that the effectiveness of OAKDA was not shown to be higher than Google results for the experiment query set.

D. DISCUSSION

The strongest performance indicator favoring OAKDA over Google was the participants own subjective rating. This may be due to test group participants feeling that they had more confirmation of the answer's correctness. Since the ontologies visually show relationships between domain concepts, the participants got affirmation more

quickly compared to reading through the prose contained in Web pages and "snippets" retrieved by Google.

The "Answer Precision" scores show OAKDA users performing better than those using Google as far as retrieving the answer text to the search question. They do not however capture the confidence level of whether the user actually believed they found the correct answer. Several query results with a high precision score surprisingly did not contain the answer text in the formulated search term list. Rather, by using the closely related, but not necessarily correct, terms in the web query, the search retrieved highly relevant hits. This result indicates that using the domain concepts which are closely related to the sought after term is a good method of formulating a Web search query. In this study it was not known how many of these high performing queries were serendipitous.

The "Context Precision" data shows that OAKDA did not deliver a performance advantage over the Google users. This result is not surprising since the keywords in the text of the search tasks contained broad contextual information about the knowledge domain. This was a conscious design choice to help the participants easily locate the appropriate ontology for the search task. The contextual information given away in the search task allowed the Google group to create queries that performed as well as the OAKDA generated queries in terms of contextual accuracy.

This results of the study demonstrate that an ontology assisted search application does indeed increase the effectiveness of the obtained results for those queries that require a deep domain knowledge, and provided that: (1) the search task's domain correlates strongly with an existing and available ontology and (2) the information the researcher starts with is sufficient to retrieve the helping ontology but is insufficient to retrieve the sought after answer.

In the study, every effort was made to craft search tasks that comply with the prerequisite conditions stated above. First, search tasks were complex enough so as to not "give away" important information that would enable an easy path to the answer via a standard Web search. Rather, the information contained in the search task was separated

by at least two degrees from the target answer. As an example, consider the following query: “Find the name of the brother of Charlie Brown’s Psychoanalyst”. Using the Peanuts cartoon domain, the answer, “Linus”, is two degrees away from “Charlie Brown”, who is the brother of Lucy, who in turn is Charlie Brown’s psychoanalyst. This is an example of a query that requires somewhat deep domain knowledge. Second, the test users were presumed to have an advantage as long as they managed to locate the ontology correctly mapped to the search task. Having done this successfully, they only needed to traverse the graph nodes from a starting point in the ontology to those nodes pointing to the answer. Correctly interpreting the meaning of search task and ontology concepts was also necessary for success. There were no instances where the test group had difficulty finding and selecting the appropriate ontology amongst pool of available OWL ontologies. If they had not, the retrieval precision scores would have been lower and much more uneven among the among the control group observations.

It is important to note, that some aspects of the study did create experimental noise in the data. The knowledge domains represented in ontologies could have been well known to some of the participants. The group scores might have had greater contrast if those participants selected had little or no knowledge in the domains selected for the search tasks. Also, while every participant answered an equal number of questions from test and control a group, not every question was visited an equal number of times. It is possible that this caused some skewing of the results. Furthermore, participant performance varied based on how closely they read and understood the search tasks as well as their skill at using and comprehending the OAKDA user interface.

Taken together, the significant means of the Rating scores and Answer precision imply that an ontology assisted search application can make a positive contribution to a researcher’s search effectiveness.

VII. SUMMARY, CONCLUSIONS AND LESSONS LEARNED

A. SUMMARY

The stated goal of this thesis was to build an ontology-based Web application to assist in domain knowledge discovery and improve Web search query by narrowing the scope of the returned list of results. To this end, the primary research question was, *Can an ontology-based application be built to narrow, expand, or refine Web search terms?* In addition, several secondary research questions follow:

1. *What is an appropriate approach for accessing and processing of contextual information of an OWL knowledge base?*
2. *What is the most appropriate architecture for the prototype application?*
3. *How can an ontology inference engine interface with the application?*
4. *Is there a method of visually rendering the ontologies for greater usability, navigation and comprehension of domain knowledge?*

This thesis addressed these questions using a two-step methodology. The first step was to research and make a case for the value of ontologies as a knowledge representation system that models the real world domain into classes, instances and the relationships between them. A thorough review of RDF and OWL provided the mechanism for understanding OWL-DL semantics and how they can be used to define the components of an ontology. It was necessary to fully comprehend the OWL constructs in order to overcome the common mistakes and pitfalls of ontology development. As part of the ontology development process, a methodology for developing OWL-DL ontologies was proposed and demonstrated by the construction of a sample ontology in the geography domain.

The second step was to design, develop and test an ontology-based Web application, called OAKDA, to allow users to search the ontology library and traverse the ontology graph to discover domain knowledge for finding relevant Web search terms. When OAKDA was completed, an experiment was conducted to test whether the system could improve the precision of Web search compared to using standard Web portal searching sites.

B. EVALUATION

The first major milestone of this thesis was the development of the sample ontologies. A significant effort was spent understanding the use of ontologies and learning RDF and OWL semantics. This knowledge was instrumental in proposing a development methodology that enabled the construction of a valid geography domain ontology. The final ontology was verified with Racer, an ontology inferencing engine, eliminating classification errors and inconsistencies. The learned development methods were later used to create the ontologies used to conduct experiments to measure OAKDA's performance in aiding Web search.

The second milestone was the overall design and execution of the OAKDA application. Bringing OAKDA from conceptualization to implementation involved finding workable solutions for key data processing areas, namely Description Logics reasoning, ontology query, and an intuitive GUI visualization framework to facilitate human interaction with the application. The selection of a multi-tier architecture contributed significantly to the success of the development effort. It enabled an effective prototyping methodology used throughout the development cycle of the application. It encouraged re-use of tier components as changes were made. The choice of Racer middleware was well suited as a platform for reasoning and querying of OWL-DL ontologies. Racer performed well as an ontology inferencing engine and provisioned an extensive language for query. Racer's reasoning service was capable of calculating implicit role relationships, reclassifying individuals based on their property restrictions, and correctly determined subsumption relationships between classes. Implementing TouchGraph software for visualization and navigation for OAKDA's GUI interface made an important contribution its usability from a user perspective.

The outcome of experiments conducted to test search performance between groups of people using OAKDA compared with Google showed that ontology aided search can confer an advantage. A performance comparison of OAKDA and Google showed a statistically significant and positive difference in the test group (OAKDA) results when performing certain research tasks. While the results showed the overall

contextual accuracy of retrieved web pages between the test and control groups were not significantly different, the test group performed better in retrieving the specific target information requested by the research task. Also, the participant survey showed that the test group rated OAKDA as more effective than Google. This implies that OAKDA users received more confirmation, or had a greater level of confidence, that they discovered the data they were tasked to retrieve.

It seems that OAKDA's effectiveness in Web research depends upon two factors – "aboutness"[27] and "availability" of the ontology information in its database. First, the degree of the ontology's aboutness to the domain of interest determines the relevancy of the knowledge found in the ontology. Terms with a high degree of aboutness do a better job of retrieving the body of information they reference. Although measuring aboutness of retrieved OWL-DL data is not in the scope of this project, an ontology is meant to be definitional or descriptive of a domain and should be useful in aiding web search as long as the ontology is not misleading in its content. Second, OAKDA relies on a presumption that an ontology exists in the users area of interest and that the ontology contains knowledge the researcher does not yet posses.

C. LESSONS LEARNED

In completing this thesis, several lessons were learned. The first was the realization that ontology development is a non trivial endeavor. Developing an ontology requires domain expert knowledge in addition to a complete understanding of the ontology language syntax and semantics. Although the level of domain knowledge required is contingent on the scope and level of detail of the ontology, having an in-depth knowledge of the domain concepts and their relationships allows the developer to accurately capture the semantics and relationships between classes. Due to the inherent level of latitude in representing type/class hierarchy in concept modeling, no two subject-matter experts (SMEs) will design an ontology of a given domain in an identical manner. Depending on the domain of context, there is usually a significant degree of subjectivity

[27] Aboutness is broadly defined as a degree in which a set of returned resources is "about" a particular domain of interest. For instance, "if a system determines that a document *d* is topically related (i.e. about) to query *q*, then the document is returned to the user." (Bruza et al., 2000, 1)

in specifying class definitions and relationships, with no absolute superiority of one particular schema over another. Also, inconsistencies are easily introduced when creating a class hierarchy of an ontology. Development tools, such as Protégé and Racer, help mitigate these errors by providing services to verify the validity and consistency of the ontology. By performing consistency checks on the ontology at various stages of its development, it was possible to find and correct classification errors or inconsistencies in class definitions.

A second lesson was learned during the experiment conducted to measure the effectiveness of the OAKDA application. The Swoogle[28] project, funded by the University of Maryland, Baltimore, is a vast searchable repository of indexed OWL and RDF ontologies. Swoogle uses automated Web crawling techniques similar to those employed by the major Web search portals to find available ontologies out on the Web. Swoogle differs from other search portals in that it only seeks RDF and OWL Web content. Swoogle was searched extensively for OWL ontologies to be used for the experiment, but none found to suitable because of their structural attributes, content or both. Instead, those ontologies had to be developed by the thesis authors. It may be that in some circumstances, ontologies need to be structured in particular ways suited to how they are being used.

D. FUTURE WORK

Currently the OWL database in OAKDA is too limited to provide usability outside of the bounds of experiment. In order for an application like OAKDA to have wide popularity as a research tool, there would need to be a repository of appropriately structured ontology files with an encyclopedic breadth. This scenario is at this time not likely to come about. OAKDA represents a departure from many other proposed uses of ontology data in that much previous work has been to try to leverage ontologies to enhance machine capabilities. OAKDA seeks to enhance the research facility of the human user. Future efforts that would be able to draw from the technologies exhibited in OAKDA might be graphical visualizations of ontologies geared to aid navigation to

[28] http://swoogle.umbc.edu - The first three letters of Swoogle stand for Semantic Web Ontology while the rest of the name is meant to refer to a popular Web search portal.

related information recomposed as an ontology. For example, suppose someone developed an ontology about a library catalog. A web site of the library's catalog could show a navigation graph depicting the kinds of relationships between related reference materials. Another usage might be based on an ontology that describes the relationships between class files, function points, variables, etc in a grouping of programming code. A visualization of this ontology could give a team of developers an easy way to navigate the structure of program and quickly learn its topology. Because of the regularity of computer programming syntax, the ontology might be able to be regenerated from the code base as changes are made.

E. CONCLUSIONS

OAKDA was built as project to explore the methodologies for accessing and processing OWL-DL ontologies in a framework which attempts to leverage domain data to enhance Web search queries. OAKDA successfully integrated several components capable of performing these operations. OAKDA's architecture demonstrates one of a few possible frameworks for attaining a Semantic Web enabled application. OAKDA overcame challenges in implementing a system to perform both machine processing and human interaction with Semantic data. OAKDA demonstrates the feasibility of these technologies and could serve as a baseline for other types of Semantic Web applications that require similar functionality.

The central assertion of Semantic Web is to provide a universal medium for the exchange of data where data can be shared and processed by automated tools as well as by people. This thesis showed the value of ontologies as a system for human-processable knowledge representation. Through applications like OAKDA, that employ OWL-DL ontologies for semantic processing, communities in every field of interest should be encouraged to capture their knowledge by developing ontologies. It is our hope that the research conducted here may suggest future methodologies for the realm of applications leveraging knowledge representation techniques.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX

A. ACRONYM TABLE

Acronym/Term	Description
API	Application Programming Interface
BOT	Software Robot
CSS	Cascading Style Sheets
DBMS	Database Management System
DL	Description Logics
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JDBC	Java™ Database Connectivity
JRE	Java™ Runtime Environment
KB	Knowledge Base
OAKDA	Ontology Aided Knowledge Discovery Application
ODBC	Open Database Connectivity
OWL-DL	Web Ontology Language- Description Logics
SOAP	Simple Object Access Protocol
SQL	Structure Query Language
TCP/IP	Transfer Control Protocol / Internet Protocol
UDDI	Universal Discovery Description Integration
Wiki	A website or similar online resource which allows users to add and edit content collectively
WSDL	Web Service Description Language
XML	eXtensible Markup Language

B. JAVA EXAMPLE CODE – "INTERFACING WITH THE RACER SERVER"

```
// instanciate new racer client instance
RacerClient client = new RacerClient("10.33.10.19",  // ip address
                                     8000);          // tcp port
// prepare read command for racer
String racerCommand  = "(owl-read-file \"" + "geography.owl" + "\")";

// try block for racer client commands
try
{
  // execute read racer file
  client.synchronousSend(racerCommand);
}
// catch io exceptions
catch (IOException ioe)
{
  System.out.println( "error during read of file." + ioe.getMessage()) ;
}
// catch racer exception
catch (RacerException re)
{
  System.out.println("error during read of file." + re.getMessage()) ;
}
```

Figure 1: Java code used to execute commands for Racer to read OWL-DL file

```
// instanciate new racer client instance
RacerClient client = new RacerClient("10.33.10.19",  // ip address
                                     8000);          // tcp port
// prepare read command for racer
String racerCommand  = "(owl-read-file \"" + "geography.owl" + "\")";
// try block
try
{
  // execute the racer read file command
  client.synchronousSend(racerCommand);
  // set string values used in racer queries
  String individual "georgia";                // set node name of individual
  String propertyName = "sharesBorderWith";   // set node name of propery
  String className = "state";                 // set class node name
  // get the name of Racers current A-Box or name of asserted content
  //    in the current ontology
  String aBox = client.currentABox();
  // query racer to get the list most specific class names from which
  //   "georgia" was instanciated
  String[] classes = client.mostSpecificInstantiators( individual,
                                                       aBox,
                                                     );
  // query racer retrieve all indivduals instanciated from the "state" class
  String[] individuals =  client.retrieveConceptInstances( className,
                                                           aBox,
                                                         );
  // query racer to get the list of individuals that
  // that share a border with "georgia"
  String[] RangeIndividuals = client.retrieveIndividualFillers( individal,
                                                                propertyName,
                                                                aBox
                                                              );
}
// catch exception
catch (IOException ioe)
{
  System.out.println( "error during read of file." + ioe.getMessage()) ;
}
// catch exception
catch (RacerException re)
{
  System.out.println( "error during read of file." + re.getMessage()) ;
}
```

Figure 2: Java code used to execute commands for Racer to read OWL-DL file and perform various queries which retrieve data in Java typed language structures.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Antoniou, Grigoris and Frank van Harmelen. 2004. *A Semantic Web Primer*. Cambridge, MA: The MIT Press.

Baader, Franz, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. 2003. *The Description Logic Handbook: Theory, Implementation, and Application*. Cambridge, UK: Cambridge University Press.

Bechhofer, Sean, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah McGuinness, Peter Patel-Schneider, and Lynn Andrea Stein. 2004. "OWL Web Ontology Language Reference." W3C Recommendation, February 2004. Available at http://www.w3.org/TR/owl-ref/. May 2005.

Beckett, Dave, ed. 2004. "RDF/XML Syntax Specification (Revised)." W3C Recommendation, February 2004. Available at http://www.w3.org/TR/rdf-syntax-grammar/. May 2005.

Berners-Lee, Tim, James Hendler, and Ora Lassila. 2001. "The Semantic Web." *The Scientific American*. Available at [http://www.scientificamerican.com/2001/0501issue/0501berners-lee.html], August 2005.

Birbeck, Mark, Jon Duckett, Oli Gauti Gudmundsson, Pete Kobak, Evan Lenz, Steve Livingstone, Daniel Marcus, Stephen Mohr, Jonathan Pinnock, Keith Visco, Andrew Watt, Kevin Williams, Zoran Zaev, and Nikola Ozu. 2001. *Professional XML, 2nd Ed*. Birmingham, UK: Wrox Press Ltd.

Brickley, Dan, and R.V. Guha, eds. 2004. "RDF Vocabulary Description Language 1.0: RDF Schema." W3C Recommendation, February 2004. Available at http://www.w3.org/TR/rdf-schema/. May 2005.

Bruza, Peter, Daiwei Song, and Kam-Fai Wong. 2000. "Aboutness from a Commonsense Perspective." Journal of the American Society of Information Science. Available at http://www.dstc.edu.au/Research/Projects/Infoeco/publications/aboutness-jasis.pdf. May 2006.

Bruza, Peter, Daiwei Song, and Kam-Fai Wong. 1999. "Fundamental Properties of Aboutness." In Proceedings of the Twenty-Second Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval (SIGIR'99), Berkeley, USA, 1999. Available at http://www.dstc.edu.au/Research/Projects/Infoeco/publications/aboutness-sigir.pdf. May 2006.

Champin, Pierre-Antoine. 2001. “RDF Tutorial.” Available at http://www710.univ-lyon1.fr/~champin/rdf-tutorial/rdf-tutorial.html. May 2006.

Chandrasekaran B, John Josephson, and V. Richard Benjamins. 1999. “What are ontologies and why do we need them?” *IEEE Intelligent Systems*. 1999;14(1):20-26. Available at http://www.infofusion.buffalo.edu/conferences_and_workshops/ontology_wkshop_2/ont_ws2_working_materials/ChandrasekaranRoleofOntology.pdf. May 2006.

Chakrabarti, Soumen. 2003. *Mining the Web: Discovering Knowledge from Hypertext Data*. San Francisco, CA: Morgan Kaufmann Publishers.

Chopra, Vivek, Ben Galbraith, Sing Li, Chanoch Wiggers, Amit Bakore, Debashish Bhattacharjee, Sandip Bhattacharya, Chad Fowler, and Romin Irani. 2003. *Professional Apache Tomcat*. Indianapolis, IN: Wiley Publishing, Inc.

Costello, Roger, and David Jacobs. 2003. “Inferring and Discovering Relationships using RDF Schemas.” Tutorial by the MITRE Corporation. Available at http://www.ics.forth.gr/isl/swprimer/presentations/rdfs.ppt. May 2006.

Daconta, Michael, Leo Obrst, and Kevin Smith. 2003. *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowledge Management*. Indianapolis, IN: Wiley Publishing, Inc.

Dameron, Oliver, Daniel Rubin, and Mark Musen. 2005. “Challenges in Converting Frame-Based Ontology into OWL: the Foundational Model of Anatomy Case-Study.” AMIA 2005 Symposium Proceedings: 181-185. Available at http://smi-web.stanford.edu/people/rubin/pubs/037_58813.pdf. May 2006.

Davies, John, Dieter Fensel, and Frank van Harmelen. 2003. *Towards the Semantic Web: Ontology-Driven Knowledge Management*. West Sussex, UK: John Wiley & Sons, Ltd.

Decker, Stefan, Prasenjit Mitra, and Sergey Melnik. 2000. “Framework for the Semantic Web: An RDF Tutorial.” In IEEE Internet Computing, November 2000. Available at http://dme.uma.pt/jcardoso/Teaching/SemanticWeb/Papers/Framework_for_the_Semantic_Web_An_RDF_Tutorial.pdf. May 2006.

Deitel, Harvey, and P.J. Deitel. 1999. *Java™: How to Program – Covers Java 2 Introducing Swing*. Upper Saddle River, NJ: Prentice Hall.

Deitel, Harvey, P.J. Deitel, T.R. Nieto, T.M. Lin, and P. Sadhu. 2001. *XML: How to Program – Featuring Java™2, Perl/CGI and Active Server Pages*. Upper Saddle River, NJ: Prentice Hall.

Doyle, Jon, and Ramesh Patil. 1991. “Two Theses of Knowledge Representation.” *Artificial Intelligence*, 48(3):261-297. 1991. Available at http://portal.acm.org/citation.cfm?id=114420&dl=GUIDE&coll=GUIDE. May 2006.

Geroimenko, Vladimir, and Chaomei Chen, eds. 2003. *Visualizing the Semantic Web: XML-based Internet and Information Visualization*. London, UK: Springer-Verlag London, LTD.

Gruber, Thomas. 1993. “A translation approach to portable ontologies.” *Knowledge Acquisition*, 5(2):199-220.

Gruber, Thomas. 1991. “The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases.” In *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, ed. J.A. Allen, R. Fikes, and E. Sandewall. San Mateo, CA: Morgan Kaufmann.

Gruininger, Michael, and Mark Fox. 1995. “Methodology for the Design and Evaluation of Ontologies.” In *Proceeding of the Workshop on Basic Ontological Issues in Knowledge Sharing, IJCAI-95*. Montreal.

Guarino, Nicola and Pierdaniele Giaretta. 1998. “Ontologies and Knowledge Bases: Towards a Terminological Clarification.” Available at http://www.loa-cnr.it/Papers/KBKS95.pdf. May 2006.

Hayes, Patrick, ed. 2004. “RDF Semantics.” W3C Recommendation, February 2004. Available at http://www.w3.org/TR/rdf-mt/. May 2005.

Haarslev, Volker and M¨oller, Ralf. 2003. "RACER User’s Guide and Reference Manual Version 1.7.7." Concordia University, Quebec, Canada University of Applied Sciences, Wedel, Germany. Available at http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-manual-1-7-7.pdf

Heaton, Jeff. 2002. *Programming Spiders, Bots, and Aggregators in Java™*. San Francisco, CA: SYBEX, Inc.

Horridge, Matthew. 2004. “A Practical Guide to Building OWL Ontologies with the Protégé-OWL Plugin,” ed. 1.0. Available at http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf. May 2006

Horrocks, Ian. 2002. “DAML+OIL: A Description Logic for the Semantic Web.” IEEE Intelligent Systems. Trends and Controversies.

Horton, Ivor. 2002. *Beginning Java 2, SDK 1.4 Ed*. Birmingham, UK: Wrox Press Ltd.

Hunter, David, Kurt Cagle, Chris Dix, Roger Kovack, Jonathan Pinnock, and Jeff Rafter. 2001. *Beginning XML, 2nd Ed*. Birmingham, UK: Wrox Press Ltd.

Israel, Saul, Norma Roemer, and Loyal Durand, Jr. 1962. *World Geography Today*. New York, NY: Holt, Rinehart and Winston, Inc.

Jasper, Robert, and Mike Uschold. 1999. "A Framework for Understanding and Classifying Ontology Applications." Boeing Math and Computing Technology.

Jones, Dean, Trevor Bench-Capon, and Pepjin Visser. 1998. "Methodologies for Ontology Development." In Proc. IT&KNOWS Conference, XV IFIP World Computer Congress, Budapest, August 1998. Available at http://www.iet.com/Projects/RKF/SME/methodologies-for-ontology-development.pdf. May 2006.

Klein, Michel, and Natalya Noy. 2003. "A Component-Based Framework for Ontology Evolution." Technical Report IR-504, Department of Computer Science, Vrije Universiteit Amsterdam, March 2003. Available at http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS//Vol-71/Klein.pdf. May 2006.

Larman, Craig. 2002. *Applying UML and Patters: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, NJ: Prentice Hall PTR.

Luke, Sean, Lee Spector, David Rager, and James Handler. 1997. "Ontology-based Web Agents." In Proceedings of the First International Conference on Autonomous Agents (Agents'97), ed. W. Lewis Johnson and Barbara Hayes-Roth, 59-68. Marina del Rey, CA: ACM Press.

Manola, Frank, and Eric Miller, eds. 2004. "RDF Primer." W3C Recommendation, February 2004. Available at http://www.w3.org/TR/rdf-primer/. May 2005.

McGuinness, Deborah. 2002. "Ontologies Come of Age." In Dieter Fensel, J im Hendler, Henry Lieberman, and Wolfgang Wahlster, eds. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2002. Available at http://www-ksl.stanford.edu/people/dlm/papers/ontologies-come-of-age-mit-press-(with-citation).htm. May 2006.

McGuinness, Deborah, and Frank van Harmelen, eds. 2004. "OWL Web Ontology Language Overview." W3C Recommendation, February 2004. Available at http://www.w3.org/TR/owl-features/. May 2005.

McLaughlin, Brett. 2001. *Java™ & XML, 2nd Ed*. Sebastopol, CA: O'Reilly & Associates, Inc.

McKinney, Kevin. 1993. *Everyday Geography: A Concise, Entertaining Review of Essential Information about the World We Live in*. Chicago, IL: Byron Press Visual Publications, Inc.

Miller, Eric. 1998. "An Introduction to the Resource Description Framework." In D-Lib Magazine, May 1998. Available at http://www.dlib.org/dlib/may98/miller/05miller.html. May 2006.

Musciano, Chuck, and Bill Kennedy. 2002. *HTML and XHTML: The Definitive Guide, 5th Ed*. Sebastolpol, CA: O'Reilly & Associates, Inc.

Noy, Natalya, and Michel Klein. 2003. "Ontology Evolution: Not the Same as Schema Evolution." Knowledge and Information Systems, 6, 2004: 428-440. Available at http://smi-web.stanford.edu/pubs/SMI_Reports/SMI-2002-0926.pdf. May 2006.

Noy, Natalya, and Deborah McGuinness. 2002. "Ontology Development 101: A Guide to Creating Your First Ontology." Available at http://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html. May 2006.

Noy, Natalya, Michael Sintek, Stefan Decker, Monica Crubezy, Ray Fergerson and Mark Musen. 2001. "Creating Semantic Web Contents with Protégé-2000." In *IEEE Intelligent Systems*. Available at http://hcs.science.uva.nl/Capita-AI/2002/papers/Noy.pdf. May 2006.

Palmer, Sean. 2001. "The Semantic Web: Introduction." Available at [http://infomesh.net/2001/swintro/]. Aug 2005.

Patel-Schneider, Peter, Patrick Hayes, and Ian Horrocks, eds. 2004. "OWL Web Ontology Language Semantics and Abstract Syntax." W3C Recommendation, February 2004. Available at http://www.w3.org/TR/owl-ref/. May 2005.

Potts, Stephen, and Mike Kopack. 2003. *Sams Teach Yourself Web Services in 24 Hours*. Indianapolis, IN: Sams Publishing.

Powers, Shelley. 2003. *Practical RDF*. Sebastopol, CA: O'Reilly & Associates, Inc.

Salkind, Neil. 2004. *Statistics for People Who Hate Statistics.* Thousand Oaks, CA: Sage Publications, Inc.

Smith, Michael, Chris Welty, and Deborah McGuinness, eds. 2004. "OWL Web Ontology Language Guide." W3C Recommendation, February 2004. Available at http://www.w3.org/TR/owl-guide/. May 2005.

Snoussi, Hicham, Laurent Magnin, and Jian-Yun Nie. 2002. “Toward an Ontology-based Web Data Extraction.” University of Montreal, Canada. Available at http://www.cs.unb.ca/ai2002/baseweb/BASeWEB2002_Paper3.pdf. May 2006.

Song, Daiwei, Kam-Fai Wong, Peter Bruza, and C.H. Cheng. 2000. “Towards A Commonsense Aboutness Theory for Information Retrieval Modeling.” University of Queensland, Australia. Available at http://www.dstc.edu.au/Research/Projects/Infoeco/publications/aboutness-sci00.pdf. May 2006.

Steelman, Andrea, and Joel Murach. 2003. *Murach’s Java Servlets and JSP*. Fresno, CA: Mike Murach & Associates, Inc.

Suryanto, Hendra, and Paul Compton. 2000. “Discovery of Ontologies from Knowledge Bases.” University of New South Wales, Australia. ECAI’2000 Workshop on Ontology Learning. Available at http://portal.acm.org/citation.cfm?id=500764&dl=GUIDE&coll=GUIDE&CFID=68130336&CFTOKEN=7204121. May 2006.

Uschold, Mike, and Michael Gruninger. 1996. “Ontologies: Principles, Methods, and Applications,” *Knowledge Engineering Review*, vol. 11, no. 2.

Wilton, Paul. 2004. *Beginning JavaScript™, 2nd Ed*. Indianapolis, IN: Wiley Publishing, Inc.

York. Richard. 2005. *Beginning CSS: Cascading Style Sheets for Web Design*. Indianapolis, IN: Wiley Publishing, Inc.

Zhang, Yi, Wamberto Vasconcelos, and Derek Sleeman. 2004. “OntoSearch: An Ontology Search Engine.” University of Aberdeen, Scotland, UK. Available at http://www.csd.abdn.ac.uk/~yzhang/AI-2004.pdf

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Harold Millan
   Defense Manpower Data Center
   harold.millan@osd.pentagon.mil